RL-TR-97-232
Final Technical Report
February 1998

SYNTHESIS APPROACH TO PARALLEL SOFTWARE ENGINEERING

Kestrel Institute

Douglas R. Smith, Thomas Emerson, LiMei Gilham, and Stephen J. Westfold

AIR FORCE RESEARCH LABORATORY
ROME RESEARCH SITE
ROME, NEW YORK

APPROVED: Joseph P Cavano

JOSEPH P. CAVANO
Project Engineer

FOR THE DIRECTOR: [illegible]

WARREN H. DEBANY JR., Technical Advisor
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE February 1998	3. REPORT TYPE AND DATES COVERED Final Sep 94 - Apr 97

4. TITLE AND SUBTITLE SYNTHESIS APPROACH TO PARALLEL SOFTWARE ENGINEERING	5. FUNDING NUMBERS C - F30602-94-C-0267 PE - 62702F PR - 5581 TA - 20 WU - 80
6. AUTHOR(S) Douglas R. Smith, Thomas Emerson, LiMei Gilham, and Stephen J. Westfold	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Kestrel Institute 3260 Hillview Avenue Palo Alto CA 94304	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTB 525 Brooks Road Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-232

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Joseph P. Cavano/IFTB/(315) 330-4033

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes our research on a synthesis approach to parallel Software Engineering. The main goal of this project was to develop concepts and generic tools to support the synthesis of parallel algorithms from formal specifications, and to carry out a representative sample of derivations for a variety of applications. Our technical approach is based on program transformation technology which allows the systematic machine-supported development of software from requirement specifications. The development process can produce highly efficient parallel code along with a proof of the code's correctness. We focused on three application domains to test our concepts.

14. SUBJECT TERMS Parallel Software Engineering, Algorithm Design, Program Synthesis, Formal Specification, Scheduling, Real-Time Tasks	15. NUMBER OF PAGES 56
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

DTIC QUALITY INSPECTED 3

Contents

List of Figures

1. Executive Summary

This report describes our research on a synthesis approach to parallel Software Engineering. The main goal of this project was to develop concepts and generic tools to support the synthesis of parallel algorithms from formal specifications, and to carry out a representative sample of derivations for a variety of applications. Our technical approach is based on *program transformation* technology which allows the systematic machine-supported development of software from requirement specifications. The development process can produce highly efficient parallel code along with a proof of the code's correctness.

Our approach to developing parallel software involves several stages. The first step is to develop a formal model of the problem domain, called a *domain theory.* Second, the constraints of a particular parallel problem are stated within a domain theory as a *problem specification.* Finally, an algorithm is produced semi-automatically by applying a sequence of *design tactics* and *program transformations* to the problem specification. The tactics and transformations embody programming knowledge about algorithms, data structures, program optimization techniques, etc. The result of the transformation process is executable code that is consistent with the problem specification.

We focused on three application domains to test our concepts:

- *Batcher's Sort* – We developed a tactic for designing divide-and-conquer algorithms in the Specware system. We used this tactic to design a well-known sorting algorithm, called Batcher's Sort, which is the most commonly used parallel sorting algorithm in current practice. Our implementation in Specware used concepts of unskolemization and ladder construction that arose from earlier experience with the KIDS system.
- *Scheduling Real-Time Concurrent Processes on Parallel Processors* – Flight avionics is an example of a critical Air Force software application that has at its core the scheduling of real-time concurrent processes on parallel processors. We exploited our experience with high-performance scheduling algorithms to synthesize a algorithm for solving this generic problem. The algorithm itself is sequential (although see the next item) but it generates a schedule of activities running in a parallel environment so that all hard deadlines and periodicity constraints are met.
- *Limited Discrepancy Search (LDS)* – LDS is a generic tree search strategy developed at University of Oregon that aims to quickly find near-optimal solutions. It has the advantage that it can also be adapted to searching a tree in parallel. We implemented LDS in KIDS as a global search program scheme. As an application, we derived a parallel version of the scheduler in the previous item – thus we have a parallel scheduler of parallel processes!

2. Introduction

Advances in parallel hardware technology have far outstripped parallel software development technology. As a result, construction of parallel programs still is more of an art than science. The increase in computing power has to be supplemented by design methods which enable us to structure problems into a form suitable for parallel computation.

We argue here for a refinement approach to developing parallel software from formal specifications of software system requirements. A refinement approach allows automated support which will in turn be necessary for constructing large-scale complex parallel software applications.

We first describe the refinement approach and then list various difficulties with current approaches to parallel software engineering and how the refinement approach addresses them. The following steps describe a refinement approach to developing a system to run on a parallel/distributed platform.

1. *Domain modeling and acquisition of a requirements specification*

   A user gathers and formulates requirements on the functionality, performance, constraints on the target language and architecture, and other properties of the desired software. The resulting specification is a pre-implementation statement of the nature of the software system and some constraints on how it is to be implemented and on what target platforms. It is intended to convey as few implementation commitments as possible, giving the maximum flexibility for choosing software architectures, algorithms, data structures, and target hardware architectures.

2. *Initial design and refinement*

   During the design phase, the user exploits various taxonomies of design knowledge: taxonomies of software architectures, algorithms [25], datatype theories, etc. A process of *classification* incrementally helps the user to discern the intrinsic structure of the specified problem [23, 22]. Exploiting this structure is the key to a well-designed system. The result of this stage is a high-level design for the system with maximal parallelism.

3. *Refinement towards particular target architectures*

   The user then begins to refine the system design towards a particular target architecture (or family of architectures). We envision a taxonomy of parallel architectures that can be exploited to incrementally refine high-level designs. The translation of a high-level design to an architecture may require the aggregation of virtual processes and optimization of the result; also, further committment to the architecture's topology affects the system's data access and communication paths. Each architecture may also support some optimizations that are not supported by its ancestor in the taxonomy, so optimizations are performed incrementally along the way. Each path in the taxonomy results in code for a different architecture. The result is a *family* of implementations, each stemming from (and consistent with) the same original specification, but each adapted to and optimized for its own architecture.

4. *System Evolution*

   Any change in the user's requirements is reflected in a change to the original specification. This change can then be propagated through the family tree of implementations, either by reusing the old derivations, or, if the change is radical enough, resynthesizing the codes with new design and optimization decisions at many points. Such evolution is inevitable and must be supported.

Machine support for this refinement process is critical because of the tremendous amount of logical detail involved at each step. Fortunately, good progress has been made on the formal foundations of the refinement approach, thereby enabling such mechanization. The many derivations of sequential programs performed using the KIDS system [24] provides some evidence for the ultimate feasibility of the refinement approach.

We now list several aspects of parallel software engineering and how the refinement approach addresses them.

1. *Reengineering of Legacy Codes*

   A considerable amount of effort has gone into tools for reengineering legacy sequential programs for vector processors. There has been limited success, but in general most old codes use inherently sequential algorithms, so reengineering can be more trouble than it is worth—it cannot fully exploit the potential parallelism, because it is not even implicitly present in the legacy code. Also, it is difficult for a reengineering approach to exploit the progress made in developing new parallel algorithms for a variety of basic problems in image processing and scientific computing, for example.

   Many contemporary parallel programming languages are extensions to established sequential languages (e.g. FORTRAN-90, CM-LISP) and are usually designed with a particular architecture in mind. Many "parallel programs" are adaptations of previously written sequential programs, especially in the scientific programming world. The focus of parallelization efforts in numerical computing has been the optimization of inner loops and the identification of idiomatic patterns of computation which can be replaced by vector operations.

   Part of the goal of parallel software engineering is to develop a methodology and tool-support that allows truly portable (or architecture-independent) program designs. Sequential code should emerge from such designs as a special case (i.e., executable code for both parallel and sequential architectures should be emittable from translators/compilers). We believe that a programming language (in which the programmer lays out an executable design in some language) is almost always at too low a level to truly attain the desired goal.

   Only by pulling back to the requirements level can we have information about the desired codes that is truly architecture-independent. It is at this abstract level that we can concentrate on the essence of the problem and expose the parallelism which is inherent. Design at this level lays out the essential computation that is required to solve the problem. The design need not concern itself with the constraints imposed by a particular architecture, e.g., boundary conditions, communication topology, etc. These issues are the subject of the rest of the development process, whose task is to realize an abstract computation on a concrete architecture.

   In short, we believe that the reengineering of legacy systems is an inherently flawed approach because the desired parallelism is usually not even implicitly present in the code. Truly portable designs will only emerge by (1) starting with architecture-independent requirement specifications, then (2) creating a high-level design with maximal parallelism, then (3) refining the design towards various target architectures.

2. *Difficulty of Writing Correct Parallel Codes*

Parallel code tends to be much more complex to understand and make correct than sequential code. One reason for this phenomenon is that there are only a few control regimes for sequential computation: if-then-else, while-do, repeat-until, etc. These idioms are well understood, and the process of putting together these idioms to produce larger programs (program synthesis and transformation) is beginning to be understood.

For parallel computation, there is more variability in control regimes because they have to be connected to the topology of the underlying architecture. Moreover, the proximity of data to processors which require it, and the resultant communication costs, are issues which have to be considered. Current models of parallel computation do not adequately capture all of these factors, thus making parallel programming difficult.

Even sequential code is a complex composition of information about the application domain, the software requirements, software architecture, algorithms, data structures, optimization techniques and details of the target platform. The extra complexity of a parallel architecture further compounds the problem. The refinement approach provides a design process that systematically adds implementation detail to a specification in a way that preserves consistency, so that *the final code is consistent with the initial specification.* Furthermore, we have seen examples of machine-generated codes that begin to press the limits of what human programmers can comprehend, and we expect more in the future. Also, the refinement steps provide a formal explanation of the code that is useful for subsequent evolutionary steps.

3. *Many Abstract Models of Parallel Computation*

   Programming is usually done in terms of a model that abstracts certain details of the underlying hardware environment. In the parallel domain, there is a diversity of parallel architectures. The most common theoretical abstract model of parallel computation is the PRAM. Unfortunately, PRAM programs tend to perform poorly on existing architectures when translated straightforwardly.

   There are abstract models that translate well for various classes of architectures, but using these models amounts to a premature commitment to the target architecture (and there is the problem of portability). So there's a tradeoff between simple abstract models of parallelism and performance on particular architectures. The natural tendency in practice is for programmers (and theoreticians) to specialize along architectural lines. It may be that there is no abstract model of parallel computation that can be automatically compiled onto all parallel architectures with acceptable efficiency.

   The refinement approach outlined above finesses this problem by exploiting a taxonomy of models (architectures) of parallel computation. Models that are deeper in the taxonomy correspond to a smaller class of target architectures and thus they can be more richly structured (in terms of processor power and structure, topology and communication costs). The refinement process can exploit the taxonomy to incrementally add detail about the target architecture to the design.

In summary, parallel software is much more difficult to construct than sequential software. Automated support will be necessary for constructing large-scale complex parallel software applications. We believe that software synthesis will prove to be the most economically viable approach to parallel software engineering.

In this project, we explored various aspects of the refinement model and carried out several applications using it. In the following sections we first introduce the KIDS system (Section 3) and our theoretical model of algorithm design (Section 4). We then describe in detail three parallel applications that we generated using KIDS or Specware.

In Section 5 we describe a tactic for designing divide-and-conquer algorithms in the Specware system. We used this tactic to design a well-known sorting algorithm, called Batcher's Sort, which is the most commonly used parallel sorting algorithm in current practice. Our implementation in Specware used concepts of unskolemization and ladder construction that arose from earlier experience with the KIDS system.

In Section 6, we describe the synthesis a algorithm for scheduling of real-time concurrent processes on parallel processors. The algorithm schedules a set of activities running in a parallel environment so that all hard deadlines and periodicity constraints are met.

In Section 7, we describe the synthesis of tree search algorithms using a novel search strategy called Limited Discrepancy Search (LDS) which was developed at University of Oregon.

3. KIDS model of program development

KIDS is a program transformation system – one applies a sequence of consistency-preserving transformations to an initial specification and achieves a correct and hopefully efficient program [24]. The system emphasizes the application of complex high-level transformations that perform significant and meaningful actions. From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in context". We hope that decisions at this level will be both intuitive to the user and be high-level enough that useful programs can be derived within a reasonable number of steps.

The user typically goes through the following steps in using KIDS for program development.

1. *Develop a domain theory* – An application domain is modeled by a domain theory (a collection of types, operations, laws, and inference rules). The domain theory specifies the concepts, operations, and relationships that characterize the application and supports reasoning about the domain via a deductive inference system. Our experience has been that distributive and monotonicity laws provide most of the laws that are needed to support design and optimization of code. KIDS has a theory development component that supports the automated derivation of various kinds of laws.

2. *Create a specification* – The user enters a problem specification stated in terms of the underlying domain theory.

3. *Apply a design tactic* – The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction (reducing a specification to a library routine), divide-and-conquer, global search (binary search, backtrack, branch-and-bound), constraint propagation, problem reduction generators (dynamic programming, general branch-and-bound, and game-tree search algorithms), and local search (hillclimbing algorithms).

4. *Apply optimizations* – The KIDS system allows the application of optimization techniques such as expression simplification, partial evaluation, finite differencing, case analysis, and

other transformations. The user selects an optimization method from a menu and applies it by pointing at a program expression. Each of the optimization methods are fully automatic and, with the exception of simplification (which is arbitrarily hard), take only a few seconds.

5. *Apply data type refinements* – The user can select implementations for the high-level data types in the program. Data type refinement rules carry out the details of constructing the implementation.

6. *Compile* – The resulting code is compiled to executable form. In a sense, KIDS can be regarded as a front-end to a conventional compiler.

Actually, the user is free to apply any subset of the KIDS operations in any order – the above sequence is typical of our experiments in algorithm design.

4. Algorithm Design and Parallelism

4.1. Problem Theories

We briefly review some basic concepts from algebra and logic. A *theory* is a structure $\langle S, \Sigma, A\rangle$ consisting of a set of sort symbols S, operations over those sorts Σ, and axioms A to constrain the meaning of the operations. A *theory morphism* (*theory interpretation*) maps from the sorts and operations of one theory to the sorts and expressions over the operations of another theory such that the image of each source theory axiom is valid in the target theory. A *parameterized theory* has formal parameters that are themselves theories [10]. The binding of actual values to formal parameters is accomplished by a theory morphism. Theory $\mathcal{T}_2 = \langle S_2, \Sigma_2, A_2\rangle$ *extends* (or is an *extension* of) theory $\mathcal{T}_1 = \langle S_1, \Sigma_1, A_1\rangle$ if $S_1 \subseteq S_2$, $\Sigma_1 \subseteq \Sigma_2$, and $A_1 \subseteq A_2$.

Problem theories define a problem by specifying a domain of problem instances or inputs and the notion of what constitutes a solution to a given problem instance. Formally, a *problem theory* $\mathcal{B}$ has the following structure.

Sorts	D, R
Operations	$I : D \to Boolean$
	$O : D \times R \to Boolean$

The *input condition* $I(x)$ constrains the input domain D. The *output condition* $O(x, z)$ describes the conditions under which output domain value $z \in R$ is a *feasible solution* with respect to input $x \in D$. Theories of booleans and sets are implicitly imported. Problems of finding optimal feasible solutions can be treated as extensions of problem theory by adding a cost domain, cost function, and ordering on the cost domain.

4.2. Algorithm Theories

An *algorithm theory* represents the essential structure of a certain class of algorithms A [25]. Algorithm theory $\mathcal{A}$ extends problem theory $\mathcal{B}$ with any additional sorts, operators, and axioms needed to support the correct construction of an $\mathcal{A}$ algorithm for $\mathcal{B}$. A theory morphism from the algorithm theory into some problem domain theory provides the problem-specific concepts needed to construct an instance of an $\mathcal{A}$ algorithm.

For example, global search theory (presented below in Section 5.3.2.) extends problem theory with the basic concepts of backtracking: subspace descriptors, initial space, the splitting and extraction operations, filters, and so on. A divide-and-conquer theory would extend problem theory with concepts such as decomposition operators and composition operators [18, 21].

The key observation in this context is that these basic algorithmic concepts are naturally parallel. Divide-and-conquer algorithms work by decomposing a hard problem instance into subproblem instances that can be solved independently (and thus in parallel). Global search algorithms work by exploring a tree of alternative solution paths – again the tree search can be easily partitioned over various processors. Other algorithm paradigms that we have studied are also naturally parallel.

5. Scheduling Parallel Tasks with Dependencies in Hard-Real-Time Systems

5.1. Introduction

In this section we consider the problem of scheduling real-time tasks to run in parallel on a pool of processors. In particular, we focus on the class of such problems encountered in "hard-real-time" systems; that is, those in which each given task has a completion deadline that must be stringently enforced if the system is to function correctly. One example of such a "hard-real-time" system is given by the operational flight program of an avionics computer with multiple CPUs, which must perform such tasks as measuring airspeed and altitude periodically. This is in contrast to the type of scheduling done, for instance, by the task dispatcher of a multiprocessor operating system, where the arrival times and durations of tasks are unpredictable, and where there may be inter-task dependencies but no real-time deadlines. Our derived algorithm performs "pre-run-time" scheduling, in which all the tasks that must be executed within a given time frame are known in advance, together with the amount of execution time that will be required by each task (or at least an upper bound if this time cannot be predicted exactly). It is common practice to use run-time (dispatch) scheduling algorithms for such problems; however, as Xu and Parnas note in their comparison of various approaches to this problem,

> [f]or satisfying timing constraints in hard-real-time systems, predictability of the system's behaviour is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system. [29]

Scheduling problems of this type are typically NP-hard ([9], A5.2). A wide variety of strategies have been used in algorithms for such problems, including branch-and-bound search [28], heuristic algorithms [30], approximation methods [11], and constraint-based temporal reasoning [3].

Our approach was to treat this problem as a special case of a more general class of problems: namely, that of scheduling a type of resources that we call *Asynchronously Shared Resources* (ASRs). An ASR is a resource that can be shared simultaneously by many users or tasks whose usage patterns are not necessarily synchronized. The resource is assumed to have finite capacity and the tasks are assumed to use the resource for finite periods. A typical example of an ASR is an automobile parking lot having n parking slots. Users can come and go independently, but a scheduler should never assign more than n users to the parking lot at the same time. More generally any pool of individual resources can be treated as an ASR: ramp space at airports, machining tools in manufacturing, computer processors running in parallel, fleets of transportation vehicles, personnel in a skill pool, etc. Note that the concept of an ASR is more general than that of a pool of resources typically considered in scheduling problems: for instance, power sources (e.g. generators, batteries) provide examples of nondiscrete ASRs.

There are several novel aspects to this work. To our knowledge we present the first solution to ASR scheduling that uses constraint propagation over time windows/bounds (however see [15] for a similar solution to the special case of a single/unshard resource). We present several solutions, experimental data, and comparative analysis. We found that the data structures necessary to support our approach are complex and the propagation control mechanisms are complex. Finally, the scheduling and constraint propagation algorithms were machine generated.

5.2. The Parallel Task Scheduling Problem

5.2.1. Homogeneous processors

Suppose we are given a set of tasks to be scheduled for execution on a pool of homogeneous processors (that is, they are identical as far as characteristics which might affect the schedule are concerned). For each task we are given an earliest start time, a latest start time, a duration, and a demand. If *tsk* denotes a task, then let *tsk.est, tsk.lst, tsk.dur, tsk.demand* denote the earliest start time, latest start time, duration, and demand, respectively, of *tsk*. Furthermore, let *tsk.eft* and *tsk.lft* denote the corresponding finish times (where $tsk.eft = tsk.est + tsk.dur$, and so on). If we regard the pool of processors as an Asynchronously Shared Resource, the scheduling problem can be formulated as follows: given

1. a set T of tasks
2. a precedence relation $\preceq$ over T (a partial order)
3. an ASR with capacity c

find an assignment of start times to each task that satisfies

1. *Precedence Constraint* – whenever task a precedes b (written $a \preceq b$) then $a.ft \leq b.st$
2. *ASR Capacity Constraint* – at no time does the demand on the ASR exceed its capacity; i.e.
$$\forall(t : time)\ \ demand(T,\ t) \leq c$$
where
$$demand(T,\ t) = \sum_{\substack{tsk \in T \\ t \in [tsk.st, tsk.ft)}} tsk.demand$$
computes the aggregate or net demand of the tasks in T at time t.

The ASR scheduling problem is easily formulated as a constraint satisfaction problem: the variables are the start times of the given tasks, and the precedence and capacity constraints restrict the possible combinations of start times that the tasks can be assigned.

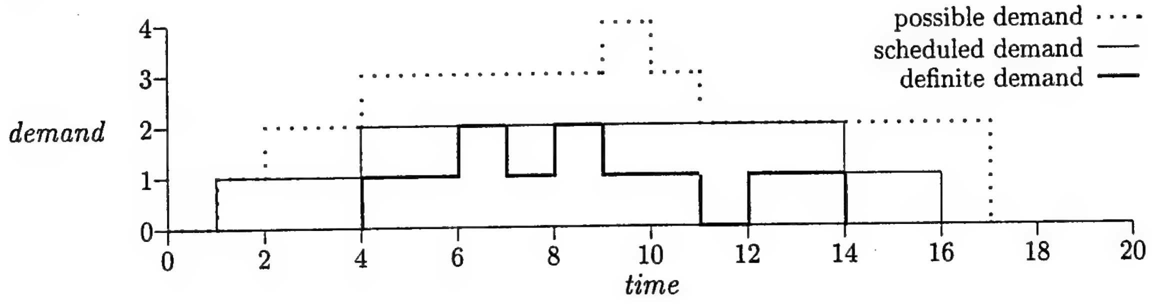

Figure 1: Demand Maps

A *schedule* is an assignment of start times to tasks. Let *tsk.st, tsk.ft* denote the start time and finish time respectively of task *tsk*. A schedule is *extractable* if the start time for each task falls within the task's *est-lst* window. An extractable schedule is *feasible* if it satisfies the precedence and capacity constraints. An *extremal extractable* schedule is obtained by systematically choosing the start time for each task to be its earliest start time (or dually latest start time).

In the parallel task scheduling problem, the capacity c is integral; and we will assume that task demands are unit (i. e., each task requires only one processor to execute). (Except for the strategy discussed in Section 5.4.1., our results readily generalize to nonintegral c and nonunit task demands.)

The aggregate demand that a set of tasks imposes on an ASR is defined in terms of actual start times for the tasks. However, during the scheduling process we only have bounds on the start times of tasks. Correspondingly, we can define bounds on the aggregate demand which we call the definite and possible demands:

$$\textit{definite–demand}(T,\ t) \ = \sum_{\substack{tsk \in T \\ t \in [tsk.lst, tsk.eft)}} tsk.demand$$

$$\textit{possible–demand}(T,\ t) \ = \sum_{\substack{tsk \in T \\ t \in [tsk.est, tsk.lft)}} tsk.demand$$

For any schedule that is extractable from a set of tasks T and any time t, we have

$$\textit{definite–demand}(T,\ t) \ \leq \ \textit{demand}(T,\ t) \ \leq \ \textit{possible–demand}(T,\ t).$$

For example, the following table specifies a set of four tasks (with no precedences) and the last column gives a feasible schedule for $c = 2$.

task	est	lst	duration	st
A	1	4	6	1
B	4	6	5	4
C	2	8	9	7
D	9	12	5	9

The corresponding demand maps are shown in Figure 1.

5.2.2. Heterogeneous Processors

In the problem considered in the previous section, we assumed that the pool of processors was homogeneous; in particular, that the duration of a task is independent of the processor on which it is executed. In this section we consider the problem in the case of heterogeneous processors, where the time actually required to execute a task may depend on which processor it is scheduled to run. Thus the problem data must now include, for each processor in the pool, the *speed* of that processor; and for each task we are given its *nominal duration*, that is, the time it would take to execute that task on a processor of speed 1.

As before, for a task *tsk*, we let *tsk.est, tsk.lft*, and *tsk.dur*, denote the earliest start time, latest finish time, and nominal duration, respectively, of *tsk*. In addition, each processor *proc* in the pool has a speed *proc.speed*. The scheduling problem for parallel tasks on heterogeneous processors can then be formulated as follows: given

1. a set T of tasks
2. a precedence relation $\preceq$ over T (a partial order)
3. a set P of k processors

find an assignment of processors and start times to tasks (that is, *tsk.st* and *tsk.proc* for each task *tsk*) that satisfies

1. *Precedence Constraint* – whenever task a precedes b (written $a \preceq b$) then
$$a.ft \leq b.st$$
where
$$a.ft = a.st + (a.dur/a.proc.speed)$$

2. *ASR Capacity Constraint* – at no time does the number of tasks being executed exceed the number of processors; i.e.
$$\forall(t : time) \;\; active(T, \; t) \leq c$$
where
$$active(T, \; t) = size(\{tsk \in T : t \in [tsk.st, tsk.ft]\})$$
is the number of tasks from T that are scheduled to execute at time t;

3. *Time Window Constraint* – for each task a
$$a.est \leq a.st$$
and
$$a.ft \leq a.lft$$

The following is an example of a parallel task scheduling problem with heterogeneous processors. As in the earlier example of scheduling homogeneous processors, we assume that there are two processors to be scheduled (which we label 1 and 2); but in this example the processors will have unequal speeds, 1.4 and 0.6, respectively. If we redefine the *capacity* c to be the sum of the speeds of the individual processors, we have $k = 2$ and $c = 2.0$. The following table shows the tasks from the previous example (with latest finish times instead of latest start times); the rightmost two columns give a feasible schedule for the execution of these tasks on this set of processors. Note that a solution now requires the assignment of an individual processor as well as a starting time to each task.

task	est	lft	duration	st	proc
A	1	10	6	1	1
B	4	11	5	6	1
C	2	17	9	2	2
D	9	17	5	10	1

Processor heterogeneity adds significantly to the complexity of the scheduling problem. Note, for instance, that there is no solution to this problem in which task A is assigned to processor 2 (for then the actual duration of A is 10, which exceeds the size of the time window for A, $A.lft - A.est$. For another example, consider the same set of tasks, but with processor speeds of 1.5 and 0.5; even though we still have $k = 2$ and $c = 2.0$, no feasible solution exists in this case.

The algorithms that were synthesized for solving these problems utilized several different search strategies. All were derived as particular instances of the general theory of global search algorithms. Before discussing the problem-specific strategies, we turn to a description of the general theory.

The problem of finding feasible schedules for a set of tasks assigned to a set of heterogeneous processors can be presented as a problem theory via a theory interpretation into the domain theory of ASR scheduling:[1]

$$\begin{array}{lll}
D & \mapsto & set(task) \times seq(processor) \\
I & \mapsto & \lambda(tasks, processors)\ true \\
R & \mapsto & map(processor, seq(reservation)) \\
O & \mapsto & \lambda(tasks, processors, sched) \\
 & & \quad Consistent\text{–}Task\text{–}Processor(sched) \\
 & & \quad \wedge\ Consistent\text{–}Earliest\text{–}Start\text{–}Times(sched) \\
 & & \quad \wedge\ Consistent\text{–}Latest\text{–}Finish\text{–}Times(sched) \\
 & & \quad \wedge\ Consistent\text{–}Task\text{–}Predecessors(sched) \\
 & & \quad \wedge\ Consistent\text{–}Task\text{–}Successors(sched) \\
 & & \quad \wedge\ Available\text{–}Processors\text{–}Used(processors, sched) \\
 & & \quad \wedge\ Scheduled\text{–}Tasks(sched) = seq\text{–}to\text{–}set(tasks)
\end{array}$$

5.3. Theory of Global Search Algorithms

5.3.1. Synthesizing a Scheduler

There are two basic approaches to computing schedules of any kind: local and global. Local methods focus on individual schedules and similarity relationships between them. Once an initial schedule is obtained, it is iteratively improved by moving to neighboring structurally similar schedules. Repair strategies [31, 14, 2, 17], fixpoint iteration [5], and linear programming algorithms are examples of local methods.

[1]The domain theory includes definitions for the types of task, processor, reservation (a record comprised of task, assigned processor, and start time) and schedule (a map from the set of processors to sequences of reservations).

Global methods focus on sets of schedules. A feasible or optimal schedule is found by repeatedly splitting an initial set of schedules into subsets until a feasible or optimal schedule can be easily extracted. Backtrack, heuristic search, and branch-and-bound methods are all examples of global methods. For this problem, we applied global methods. In the following subsections we formalize the notion of global search method and show how it can be applied to synthesize a scheduler. Other projects taking a global approach include ISIS [8], OPIS/DITOPS [27], and MicroBoss [16] (all at CMU).

5.3.2. Global Search Theory

The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets, and eliminates sets via filters until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters serve to prune off branches of the tree that cannot lead to solutions.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus global search algorithms are based on an abstract data type of intensional representations called *space descriptors* (denoted by hatted symbols). In addition to the extraction and splitting operations mentioned above, the type also includes a predicate *satisfies* that determines when a candidate solution is in the set denoted by a descriptor. Further, there is a refinement relation on spaces that corresponds to the subset relation on the sets denoted by a pair of descriptors.

The various operations in the abstract data type of space descriptors together with problem specification can be packaged together as a theory. Formally, abstract *global search theory* (or simply *gs-theory*) $\mathcal{G}$ is presented in Figure 2, where D is the input domain, R is the output domain, I is the input condition, O is the output condition, $\hat{R}$ is the type of space descriptors, $\hat{I}$ defines legal space descriptors, $\hat{r}$ and $\hat{s}$ vary over descriptors, $top(x)$ is the descriptor of the initial set of candidate solutions, $Satisfies(z, \hat{r})$ means that z is in the set denoted by descriptor $\hat{r}$ or that z satisfies the constraints that $\hat{r}$ represents, and $Extract(z, \hat{r})$ means that z is directly extractable from $\hat{r}$.

The relations *Split–Arg* and *Split–Constraint* are used to determine and perform splitting. In particular, if $Split\text{–}Arg(x, \hat{r}, c)$ then c is information that characterizes (or informs) one branch of the split. $Split\text{–}Constraint(x, \hat{r}, c, \hat{s})$ means that $\hat{s}$ results from incorporating information c into the descriptor $\hat{r}$ (with respect to input x). *Split–Arg* is used to control the generation of children of a node in the search tree and *Split–Constraint* is used to specify one child. *Split–Constraint* can be thought of as a parameterized constraint whose alternative arguments are supplied by *Split–Arg*.

The refinement relation $\hat{r} \sqsupseteq \hat{s}$ holds when $\hat{s}$ denotes a subset of the set denoted by $\hat{r}$. Further, $\hat{R}$ together with $\sqsupseteq$ forms a bounded semilattice. This structure will play a crucial role in constraint propagation algorithms.

Note that all variables in the axioms are assumed to be universally quantified unless explicitly specified otherwise. Axiom GS0 asserts that the initial descriptor $top(x)$ is a legal descriptor.

Global Search Theory

Spec *Global-Search*

Sorts	D	*input domain*
	R	*output domain*
	$\hat{R}$	*subspace descriptors*
	$\hat{C}$	*splitting information*

Operations

$I : D \rightarrow boolean$	*input condition*
$O : D \times R \rightarrow boolean$	*input/output condition*
$\hat{I} : D \times \hat{R} \rightarrow boolean$	*subspace descriptors condition*
$Satisfies : R \times \hat{R} \rightarrow boolean$	*denotation of descriptors*
$Split\text{–}Arg : D \times \hat{C} \times \hat{R} \rightarrow boolean$	*specifies arguments to split constraint*
$Split\text{–}Constraint : D \times \hat{R} \times \hat{C} \times \hat{R} \rightarrow boolean$	*parameterized splitting constraint*
$Extract : R \times \hat{R} \rightarrow boolean$	*extractor of solutions from spaces*
$\Psi : D \times R \times \hat{R} \rightarrow boolean$	*cutting constraint*
$\xi : D \times \hat{R} \rightarrow boolean$	*cutting constraint*
$\sqsupseteq : D \times \hat{R} \times \hat{R} \rightarrow boolean$	*refinement relation*
$top : D \rightarrow \hat{R}$	*initial space*
$bot : \hat{R}$	*inconsistent space*

Axioms

GS0. All feasible solutions are in the *top* space

$$I(x) \ \wedge \ O(x,z) \implies Satisfies(z, top(x))$$

GS1. All solutions in a space are finitely extractable

$$I(x) \ \wedge \ \hat{I}(x,\hat{r})$$
$$\implies (Satisfies(z,\hat{r}) \iff \exists(\hat{s}) \ (\ Split^{*}(x,\hat{r},\hat{s}) \ \wedge \ Extract(z,\hat{s})))$$

GS2. Specification of Cutting Constraint

$$Satisfies(z,\hat{r}) \ \wedge \ O(x,z) \implies \Psi(x,z,\hat{r})$$

GS3. Definition of Cutting Constraint on Spaces

$$\xi(x,\hat{r}) \iff \forall(z:R)(\ Sat(z,\hat{r}) \implies \Psi(x,z,\hat{r}))$$

GS4. Definition of Refinement

$$\hat{r} \sqsupseteq \hat{s} \iff \forall(z:R)(Satisfies(z,\hat{s}) \implies Satisfies(z,\hat{r}))$$

GS5. $\langle \hat{R}, \sqsupseteq, \sqcap, top, bot \rangle$ is a bounded meet-semilattice with *bot* as universal lower bound.

end spec

Figure 2: Global Search Theory

Axiom GS1 asserts that legal descriptors split into legal descriptors and that *Split* induces a well-founded ordering on spaces. Axiom GS2 constrains the denotation of the initial descriptor — all feasible solutions are contained in the initial space. Axiom GS3 gives the denotation of an arbitrary descriptor $\hat{r}$ — an output object z is in the set denoted by $\hat{r}$ if and only if z can be extracted after finitely many applications of *Split* to $\hat{r}$ where

$$Split^*(x, \hat{r}, \hat{s}) \iff \exists(k : Nat)\ \ Split^k(x, \hat{r}, \hat{s})$$

and

$$Split^0(x, \hat{r}, \hat{t}) \iff \hat{r} = \hat{t}$$

and for all natural numbers k

$$Split^{k+1}(x, \hat{r}, \hat{t})$$
$$\iff \exists(\hat{s} : \hat{R},\ i : \hat{C})\ (\ Split\text{–}Arg(x, \hat{r}, i)\ \wedge\ Split\text{–}Constraint(x, \hat{r}, i, \hat{s})\ \wedge\ Split^k(x, \hat{s}, \hat{t})).$$

Axiom GS4 asserts that if $\hat{r}$ splits to $\hat{s}$ then $\hat{r}$ also refines to $\hat{s}$; thus the refinement relation on $\hat{R}$ is weaker than the split relation. We also need the axioms that $\langle \hat{R}, \sqsupseteq, \sqcap \rangle$ is a semilattice. For simplicity, we write $\hat{r} \sqsupseteq \hat{s}$ rather than the correct $\sqsupseteq (x, \hat{r}, \hat{s})$; and similarly $\hat{r} \sqcap \hat{s}$.

For example, a simple global search theory of parallel task scheduling (homogeneous case) has the following form. Schedules are represented as maps from processors to sequences of reservations (where each reservation includes a task, earliest-start-time, latest-finish-time, and actual start time). The type of schedules has the invariant (or subtype characteristic) that for each reservation, the earliest-start-time plus the task duration is no later than the latest-finish-time. A partial schedule is a schedule over a subset of the given tasks.

The initial (partial) schedule is just the empty schedule – a map from the available processors to the empty sequence of reservations. A partial schedule is extended by first selecting a task, *task*, to schedule, and then selecting a processor, *proc*. The tuple $\langle task, proc \rangle$ constitutes the information c of *Split–Arg*. *Split–Constraint*, given $\langle task, proc \rangle$, creates an extended schedule that has a reservation for *tsk* added to the sequence of reservations currently scheduled on *proc*. The alternative ways that a partial schedule can be extended naturally gives rise to the branching structure underlying global search algorithms.

The formal version of this global search theory of scheduling can be inspected in the domain theory in Appendix C.

5.3.3. Pruning Mechanisms

When a partial schedule is extended it is possible that some problem constraints are violated in such a way that further extension to a complete feasible schedule is impossible. In tree search algorithms it is crucial to detect such violations as early as possible.

Pruning tests are derived in the following way. The test

$$\exists(z)\ (Satisfies(z, \hat{r})\ \wedge\ O(x, z)) \tag{1}$$

decides whether there exist any feasible solutions that are in the space denoted by $\hat{r}$. If we could decide this at each node of our branching structure then we would have perfect search – no deadend branches would ever be explored. In practice it would be impossible or horribly complex to compute

(1), so we rely instead on an inexpensive approximation to it. In fact, if we approximate (1) by weakening it (deriving a necessary condition of it) we obtain a sound pruning test. That is, suppose we can derive a test $\Phi(x, \hat{r})$ such that

$$\exists(sched)\ (Satisfies(z, \hat{r})\ \wedge\ O(x, z)) \implies \Phi(x, \hat{r}). \tag{2}$$

By the contrapositive of (2), if $\neg\Phi(x, \hat{r})$ then there are no feasible solutions in $\hat{r}$, so we can eliminate it from further processing. A global search algorithm will test Φ at each node it explores, pruning those nodes where the test fails.

More generally, necessary conditions on the existence of feasible (or optimal) solutions below a node in a branching structure underlie pruning in backtracking and the bounding and dominance tests of branch-and-bound algorithms [19].

It appears that the bottleneck analysis advocated in the constraint-directed search projects at CMU [7, 16] leads to a semantic approximation to (1), but neither a necessary nor sufficient condition. Such a *heuristic* evaluation of a node is inherently fallible, but if the approximation is close enough it can provide good search control with relatively little backtracking.

To derive pruning tests for the ASR scheduling problem, we instantiate (1) with our definition of *Satisfies* and *O* and use an inference system to derive necessary conditions. The resulting tests are fairly straightforward; of the 7 original feasibility constraints, 5 yield pruning tests on partial schedules. For example, the partial schedule must satisfy *Consistent-Task-Processor*, *Consistent-Separation-on-Processor-EST*, *Consistent-Separation-on-Processor-LFT*, *Consistent-Separation-Predecessors*, *Consistent-Separation-Scheduled-Successors*, *Consistent-Separation-Scheduled-to-Unscheduled-Successors*, and *Consistent-Separation-Unscheduled-to-Unscheduled-Successors*. The reader may note that computing these tests on partial schedules is rather expensive and mostly unnecessary; however, later program optimization steps reduce these tests to fast and irredundant form. For example, the second test will reduce to checking that, when we assign a task to processor i, the earliest start time of the newly assigned task is consistent with the earliest start time of the task on the same processor that immediately precedes it.

For details of deriving pruning mechanisms for other problems see [19, 24, 25, 20].

5.3.4. Cutting Constraints and Constraint Propagation

Constraint propagation is a more general technique that is crucial for early detection of infeasibility. We developed a general mechanism for deriving constraint propagation code and applied it to scheduling.

Each node in a backtrack tree can be viewed as a data structure that denotes a set of candidate solutions – in particular the solutions that occur in the subtree rooted at the node (see Figure 3). Thus the root denotes the set of all candidate solutions found in the tree.

Pruning has the effect of removing a node (set of solutions) from further consideration. In contrast, constraint propagation has the effect of changing the space descriptor so that it denotes a smaller set of candidate solutions. The effect of constraint propagation is to spread information through the subspace descriptor resulting in a tighter descriptor and possibly exposing infeasibility. Pruning can be treated as a special case of propagation in which a space is refined to descriptor that denotes the empty set of solutions.

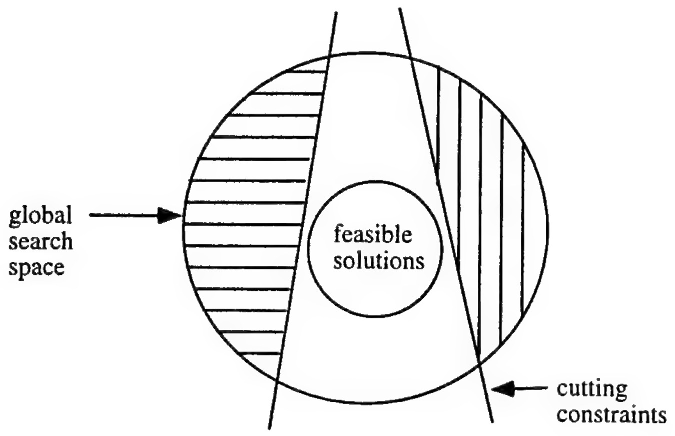

Figure 3: Global Search Subspace and Cutting Constraints

Constraint propagation is based on the notion of *cutting constraints* which are necessary conditions $\Psi(x, z, \hat{r})$ that a candidate solution z satisfying $\hat{r}$ is feasible:

$$\forall(x : D, \hat{r} : \hat{R},\ z : R)(\mathit{Satisfies}(z, \hat{r})\ \wedge\ O(x, z) \implies \Psi(x, z, \hat{r})) \tag{3}$$

See Figures 3 and 4. In order to get a test on spaces that decides whether Ψ has been incorporated, we make one further definition:

$$\xi(x, \hat{r}) \iff \forall(z : R)(\mathit{Satisfies}(z, \hat{r}) \implies \Psi(x, z, \hat{r})) \tag{4}$$

The test $\xi(x, \hat{r})$ holds exactly when all candidate solutions in $\hat{r}$ satisfy Ψ, and we say that $\hat{r}$ *satisfies* ξ.

The key question at this point is: Given a descriptor $\hat{r}$ that doesn't satisfy ξ, how can we incorporate ξ into $\hat{r}$? The answer is to find the greatest refinement of $\hat{r}$ that satisfies ξ; we say $\hat{t}$ *incorporates* ξ into $\hat{r}$ if

$$\hat{t} = max_{\sqsupseteq}\{\hat{s} \mid \hat{r} \sqsupseteq \hat{s}\ \wedge\ \xi(x, \hat{s})\}. \tag{5}$$

which asserts that $\hat{t}$ is maximal over the set of descriptors that refine $\hat{s}$ and satisfy ξ, with respect to ordering $\sqsupseteq$. We want $\hat{t}$ to be a refinement of $\hat{r}$ so that all of the information in $\hat{r}$ is preserved and we want $\hat{t}$ to be maximal so that no other information than $\hat{r}$ and ξ is incorporated into $\hat{t}$.

The next question concerns the conditions under which Formula (5) is satisfiable. Assuming that $\hat{R}$ is a semilattice, we can use variants of Tarski's fixpoint theorem (c.f. [5]):

Theorem If there is a function f such that

1. f is monotonic on $\hat{R}$ (i.e. $\hat{s} \sqsupseteq \hat{t} \implies f(x, \hat{s}) \sqsupseteq f(x, \hat{t})$)

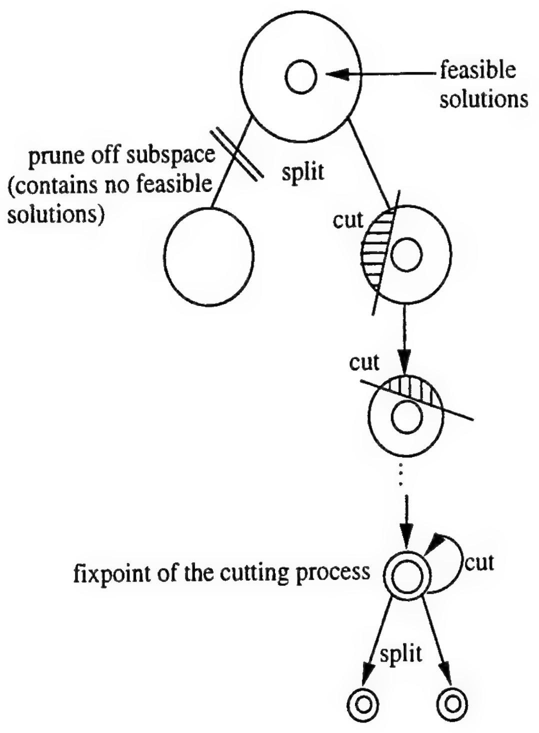
feasible solutions
prune off subspace (contains no feasible solutions)
split
cut
cut
fixpoint of the cutting process
cut
split

Figure 4: Pruning and Constraint Propagation

2. f is deflationary (i.e. $\hat{r} \sqsupseteq f(x, \hat{r})$)
3. f has fixed-points satisfying ξ (i.e. $f(x, \hat{r}) = \hat{r} \iff \xi(x, \hat{r})$)

then (1) $\hat{t} = max_{\sqsupseteq}\{\hat{s} \mid \hat{r} \sqsupseteq \hat{s} \wedge \xi(x, \hat{s})\}$ exists
and (2) $\hat{t}$ is the greatest fixpoint of f; i.e. $\hat{t}$ can be computed by iteratively applying f to $\hat{r}$ until a fixpoint is reached.

The challenge is to construct a monotonic, deflationary function whose fixed-points satisfy ξ. A general construction in terms of global search theory can be sketched as follows. Let

$$f(x, \hat{r}) = \begin{cases} \hat{r} & \textit{if } \xi(x, \hat{r}) \\ \ldots & \textit{if } \neg\xi(x, \hat{r}) \end{cases}$$

The intent is to define f so that it has fixpoints exactly when $\xi(x, \hat{r})$ holds. When $\xi(x, \hat{r})$ doesn't hold, then we know (by the definition of ξ and the contrapositive of formula (3)) that

$$\exists(z : R)(\mathit{Satisfies}(z, \hat{r}) \wedge \neg O(x, z))$$

i.e. there are some infeasible solutions in the space described by $\hat{r}$. Ideally $\neg\xi(x, \hat{r})$ is a *constructive* assertion, so it provides information on *which* solutions are infeasible and how to eliminate them. In place of the ellipsis above we require a new descriptor that refines $\hat{r}$ (so f is decreasing on all inputs), allows f to be monotone, and eliminates some of the infeasible solutions indicated by

$\neg\xi(x,\hat{r})$. In general it is difficult to see how to achieve this end without assuming special structure to $\hat{R}$ and ξ.

We have identified some special cases for which an analytic procedure can produce the necessary iteration function f from ξ. These special cases subsume our scheduling applications and many related Constraint Satisfaction Problems (CSP) problems. Suppose that the constraint ξ has the form

$$B(x,\hat{r}) \sqsupseteq \hat{r} \tag{6}$$

where $B(x,\hat{r})$ is monotonic in $\hat{r}$. We say that ξ is a *Horn-like constraint* by generalization of Horn clauses in logic. Notice that the occurrence of $\hat{r}$ on the right-hand side of the inequality has positive polarity (i.e. it is monotonic in $\hat{r}$), whereas the occurrence(s) of $\hat{r}$ on the left-hand side have negative polarity (i.e. are antimonotonic). If the constraint were boolean (with B and $\hat{r}$ being boolean values and $\sqsupseteq$ being implication), then this would be called a definite Horn clause. When our constraints are Horn-like, then there is a simple definition for the desired function f:

$$f(x,\hat{r}) = \begin{cases} \hat{r} & \textit{if } B(x,\hat{r}) \sqsupseteq \hat{r} \\ B(x,\hat{r}) \sqcap \hat{r} & \textit{if } \neg B(x,\hat{r}) \sqsupseteq \hat{r} \end{cases}$$

or equivalently

$$f(x,\hat{r}) = B(x,\hat{r}) \sqcap \hat{r}.$$

It is easy to check that f is monotone in $\hat{r}$, deflationary, and has fixed-points exactly when ξ holds. Therefore, simple iteration of f will converge to the descriptor that incorporates ξ into $\hat{r}$. However, if $\hat{r}$ is an aggregate structure such as a tuple or map, then the changes made at each iteration may be relatively sparse, so the simple iteration approach may be grossly inefficient. We found this feature to be characteristic of scheduling and other CSPs. Our approach to solving this problem is to focus on single point changes and to exploit dependence analysis. For each component of $\hat{r}$ we define a separate change propagation procedure. The arguments to a propagation procedure specify a change to the component. This change is performed and then the change procedures for all other components that could be affected by the change are invoked. Static dependence analysis at design-time is used to determine which constraints could be affected by a change to a given component.

A program scheme for global search with constraint propagation is presented in Figure 5. The global search design tactic in KIDS first instantiates this scheme, then invokes a tactic for synthesizing propagation code to satisfy the specification *F–split–and–propagate.*

CSPs with Horn-like constraints

We now elaborate the previous exposition of propagation of Horn-like constraints arising in CSPs. To keep matters simple, yet general, suppose that the output datatype R is *map(VAR, VALSET)*, where VAR is a type of variables, and $VALSET$ is a type that denotes a set of values (this implies that all the variables have the same type and refinement ordering), and the $\sqsupseteq$ relation has the form:

$$\hat{r} \sqsupseteq \hat{s} \;\; \textit{iff} \;\; \bigwedge_{v} \hat{r}(v) \sqsupseteq \hat{s}(v).$$

Global Search Program Theory

Spec *Global-Search-Program (T :: Global-Search)*

Operations

```
F-initial-propagate  $(x : D \mid I(x))$
    returns $(\hat{t} : \hat{R} \mid \hat{t} = max_{\sqsupseteq}\ \{\hat{s} \mid top(x) \sqsupseteq \hat{s}\ \wedge\ \hat{I}(x,\hat{s})\ \wedge\ \xi(x,\ \hat{s})\})$

F-split-and-propagate
    $(x : D,\ \hat{r} : \hat{R},\ c : \hat{C}$
    $\mid I(x)\ \wedge\ \hat{I}(x,\hat{r})\ \wedge\ Split\text{–}Arg(x,\hat{r},c)\ \wedge\ \xi(x,\hat{r})\ \wedge\ \hat{r} \neq bot)$
    returns $(\hat{t} : \hat{R} \mid \hat{t} = max_{\sqsupseteq}\ \{\hat{s} \mid \hat{r} \sqsupseteq \hat{s}\ \wedge\ \hat{I}(x,\hat{s})$
                                            $\wedge\ Split(x,\hat{r},c,\hat{s})\ \wedge\ \xi(x,\hat{s})\})$

F-gs $(x : D,\ \hat{r} : \hat{R}\ \mid\ I(x)\ \wedge\ \hat{I}(x,\hat{r})\ \wedge\ \Phi(x,\hat{r}))$
    returns $(z : R \mid O(x,z)\ \wedge\ Satisfies(z,\hat{r}))$
    = if $\exists(z)\ (Extract(z,\hat{r})\ \wedge\ O(x,z))$
        then $some(z)\ (Extract(z,\hat{r})\ \wedge\ O(x,z))$
        else $some(z)\ \exists(c : \hat{C},\ \hat{t} : \hat{R})$
                        $(Split\text{–}Arg(x,\hat{r},c)$
                        $\wedge\ \hat{t} = F\text{-}split\text{-}and\text{-}propagate(x,\hat{r},c)\ \wedge\ \hat{t} \neq bot$
                        $\wedge\ z = F\text{-}gs(x,\hat{t}))$

F $(x : D \mid I(x))$
    returns $(z : R \mid O(x,z))$
    = $some(z)\ \exists(\hat{t})\ \ (\hat{t} = F\text{-}initial\text{-}propagate(x)$
                        $\wedge\ \hat{t} \neq bot$
                        $\wedge\ z = F\text{-}gs(x,\hat{t}))$
```

end spec

Figure 5: Global Search Program Theory

Suppose further that ξ is a conjunction of constraints giving bounds on the variables:

$$\xi(x, \hat{r}) \iff \bigwedge_{v} B_v(x, \hat{r}) \sqsupseteq \hat{r}(v)$$

where $B_v(x, \hat{r})$ is monotonic in $\hat{r}$. Under these assumptions, $\neg\xi(x, \hat{r})$ implies that the bounding constraint on some variable v is violated; i.e.

$$\neg B_v(x, \hat{r}) \sqsupseteq \hat{r}(v).$$

To "fix" such a violation we can change the current valset of v to

$$B_v(x, \hat{r}) \sqcap \hat{r}(v),$$

which simultaneously refines $\hat{r}(v)$, since

$$\hat{r}(v) \sqsupseteq B_v(x, \hat{r}) \sqcap \hat{r}(v)$$

and reestablishes the constraint on v, since

$$B_v(x, \hat{r}) \sqsupseteq B_v(x, \hat{r}) \sqcap \hat{r}(v).$$

Let

$$B(x, \hat{r}) = \{\mid u \rightarrow B_u(x, \hat{r}) \sqcap \hat{r}(u) \mid u \in domain(\hat{r}) \mid\}$$

then, define f as:

$$f(x, \hat{r}) = \hat{r} \sqcap B(x, \hat{r})$$

Constraint propagation is treated here as iteration of f until a fixed-point is reached. Efficiency requires that we go farther, since only a sparse subset of the variables in $\hat{r}$ will be updated at each iteration. If we implemented the iteration on a vector processor or SIMD machine, the overall computation could be fast, but wasteful of processors. On a sequential machine, it is advantageous to analyze the constraints in ξ to infer dependence of constraints on variables. That is, if (the valset of) variable v changes, which constraints in ξ could become violated? This dependence analysis can be used to generate special-purpose propagation code as follows.

For each variable v, let *affects(v)* be the set of variables whose constraints could be violated by a change in v; more formally, let

$$affects(v) \;=\; \{u \mid v \text{ occurs in } B_u \}.$$

We can then generate a set of procedures that carry out the propagation/iteration of f: For each variable v, generate the following propagation procedure:

```
$Propagate_v\ (x : D,\ \hat{r} : \hat{R},\ new\text{–}valset : VALSET$
                $\mid I(x) \ \wedge\ \hat{I}(x, \hat{r})$
                $\wedge\ \hat{r}(v) \sqsupset new\text{–}valset$
                $\wedge\ B_v(x, \hat{r}) \sqsupseteq new\text{–}valset)$
= let $(\hat{s} : \hat{R} = map\text{–}shadow(\hat{r}, v, new\text{–}valset))$
    if $\neg\hat{I}(x, \hat{s})$ then $bot$
    else
    ... for each variable $u$ in $affects(v)$ ...
    ... generate the following code block ...
    if $\hat{s}$ = bot then bot
    else (if $\neg(B_u(x,\hat{s}) \sqsupseteq \hat{s}(u))$
          then $\hat{s} \leftarrow Propagate_u(x, \hat{s},\ B_u(x,\hat{s}) \sqcap \hat{s}(u)))$;
    ...
    $\hat{s}$
end
```

where *map–shadow*$(\hat{r}, v, new\text{–}valset)$ returns the map $\hat{r}$ modified so that $\hat{r}(v) = new\text{–}valset$.

To finish up, if *Split*$(x, \hat{r}, i, \hat{s})$ has the form

$$\hat{s}(u) = C(x, \hat{r}, i)$$

for some function C that yields a refined valset for variable u, then we can satisfy *F–split–and–propagate* as follows:

$$F\text{–}split\text{–}and\text{–}propagate(x, \hat{r}, i) = propagate_u(\hat{r}, C(x, \hat{r}, i)).$$

The change to u induced in the call to $propagate_u$ will in turn trigger changes to other variables, and so on.

Constraint Propagation for Parallel Task Scheduling

For parallel task scheduling, each iteration of the *Propagate* operation has the following form: For each processor *proc* let *proc*(i) be the i-th reservation on *proc*, and *proc*(i)*.task* the task for that reservation. Letting est_i denote the current value of earliest-start-time for that task and est_i' the next value of the earliest-start-time for that task (with lft_i and lft_i' defined analogously), we have

$$est_i' = max \begin{cases} est_i \\ est_{i-1} + actual\text{–}duration_{i-1} \\ \max\{tsk.est + tsk.actual\text{–}duration \mid tsk \preceq proc(i).task\} \end{cases} \tag{7}$$

$$lft_i' = min \begin{cases} lft_i \\ lft_{i+1} - actual\text{–}duration_{i+1} \\ \min\{tsk.lft - tsk.actual\text{–}duration \mid proc(i).task \preceq tsk\} \end{cases} \tag{8}$$

Here $actual\text{–}duration_{i-1}$ is the time taken to execute the *(i-1)*-th task on *proc*, i. e., $nominal - duration_i/proc.speed$. Boundary cases must be handled appropriately.

After adding a new reservation to some trip, the effect of *Propagate* will be to shrink the $\langle est,\ lft\rangle$ window of each task on the same processor, and possibly also predecessor and successor tasks of the newly aded task. (Note that propagation over successors requires that the generated propagation code must deal with the unscheduled tasks as well as the partially completed schedule, an example of propagation over heterogeneous data types). If the size of the time window becomes negative or zero for any task, the partial schedule is necessarily infeasible and it can be pruned.

5.4. Scheduling Strategies

As noted previously, we treated the parallel task scheduling problem as a special case of the ASR problem. There are a variety of strategies for solving ASRs, all based on branch-and-propagate. In each subsection below we present Horn-like constraints derived from the ASR capacity constraint and describe some data structures and control strategies that can effectively apply them.

5.4.1. Discrete strategy – scheduling individual processors

A *discrete* strategy assumes that the capacity bound on the ASR is integral and that each task consumes one unit of capacity. At each branching point, an unscheduled task is selected, assigned to one of the k processors, and finally to some position within the sequence of tasks previously assigned to that processor. The main constraint that is propagated asserts that one task must finish before the next one on the same processor can start. A data structure that represents a map from processors to sequences of reservations facilitates this strategy. The data structure forces the satisfaction of the capacity constraint by construction [2]; or, from another point of view, the residual propagation necessary to satisfy the capacity constraint is precedence between consecutive tasks for each processor.

The derived constraints are as follows. Let m be a map from $\{1..k\}$ to sequences of tasks. From the basic constraint (expressed over start times)

$$\begin{array}{l}\forall(proc : integer,\ index : integer)\\ \quad (proc \in \{1..k\}\\ \quad \wedge\ index \in \{1..size(m(proc)) - 1\}\\ \quad \Longrightarrow\\ \quad m(proc)(index).ft\ \leq\ m(proc)(index + 1)).st\end{array}$$

we can infer (a indexed collection of) Horn-like propagation constraints over start time windows:

$$\begin{array}{l}\forall(proc : integer,\ index : integer)\\ \quad (proc \in \{1..k\}\\ \quad \wedge\ index \in \{1..size(m(proc)) - 1\}\\ \quad \Longrightarrow\\ \quad m(proc)(index).eft\ \leq\ m(proc)(index + 1).est\\ \quad \wedge\ m(proc)(index).lft\ \leq\ m(proc)(index + 1).lst)\end{array}$$

The conjunct

$$m(proc)(index).eft\ \leq\ m(proc)(index + 1).est$$

provides a bound on $m(proc)(index+1).est$: whenever $m(proc)(index).eft$ increases, then the constraint may become invalid; in order to reestablish the constraint, the value of on $m(proc)(index+1).est$ can then be increased (by the minimal amount) to $m(proc)(index).eft$. Note that $m(proc)(index).eft$ is monotone under refinement, since the earliest start time of a task can only increase. Correspondingly, the second conjunct provides an upper bound on $m(proc)(index).lft$ which decreases (monotonically) whenever $m(proc)(index + 1).lst$ decreases.

[2]This, of course, is where we use the simplifying assumption that the ASR capacity is an integer

There are several advantages to the discrete approach. First, unlike the aggregate ASR scheduling strategy (discussed in 5.4.2.), the discrete strategy applies to both the homogeneous and heterogeneous variants of the problem. Second, partial schedules are clear and definite. And third, this constraint has the property that extremal extractable schedules are always feasible (relative to the tasks scheduled so far). (We have also used this approach in scheduling parking slots and ground crews in airlift scheduling in the ITAS system [4].)

The main disadvantage is that the branching factor is very high and grows as $O(kn)$ where n is the number of allocated tasks. Nevertheless, good heuristics for choosing a processor and position within the schedule for a processor can result in a fast heuristic algorithm producing good results. Our synthesized algorithm used two heuristics (incorporated into the global search theory for this strategy) to help keep the branching factor at a computationally practical level: First, the next task to be scheduled is always appended to the sequence of tasks already scheduled on a processor (instead of being inserted at some point within the sequence). Second, the processors are ordered in such a way (namely, in decreasing order of processor speed) that (other things being equal) a task to be scheduled is more likely to be assigned to a fast processor than a slow one; thus the earlier branches of the search tree tend to represent partial schedules in which the fastest processors run the most tasks.

5.4.2. Aggregate strategies

In *aggregate* strategies we treat the scheduling of an ASR as a whole, assuming no internal structure to the capacity of an ASR. In the context of the problem under consideration, this means that we do not assign a task to a specific processor when it is scheduled; thus an aggregate strategy can only be used for the homogeneous variant of the parallel task scheduling problem. At each branching point, an unscheduled task is selected, and it is added to the partial schedule constructed so far. Its addition may trigger various constraints, in particular a disjunctive constraint which effectively does the branching. Data structures that represent possible and definite demand maps facilitate this strategy. More details on aggregate strategies may be found in [26].

5.5. Experience with generated code

We used KIDS [24] to synthesize a variety of ASR scheduling algorithms, in particular two discrete algorithms called *Discrete-1* and *Discrete-2*, as described in Section 5.4.1.. They differ primarily in the variants of problems that they solve: *Discrete-1* deals with the homogeneous case with no precedence relationships, *Discrete-2* allows heterogeneous processors and precedence relationships between tasks. Two aggregate algorithms, called *Aggregate-1* and *Aggregate-2* were also synthesized. The aggregate algorithms differ in their strategies for splitting and extracting.

5.6. Timing experiments

We ran two sets of experiments with the generated code. In the first set we generated a number of random problems of the homogeneous variant with no precedence relationships between tasks, and compared the performance of the generated algorithms which solved that case only, namely *Aggregate-1*, *Aggregate-2*, and *Discrete-1*. In the second set we determined the time complexity of algorithm *Discrete-2* on heterogeneous problems with precedence constraints, using the number of

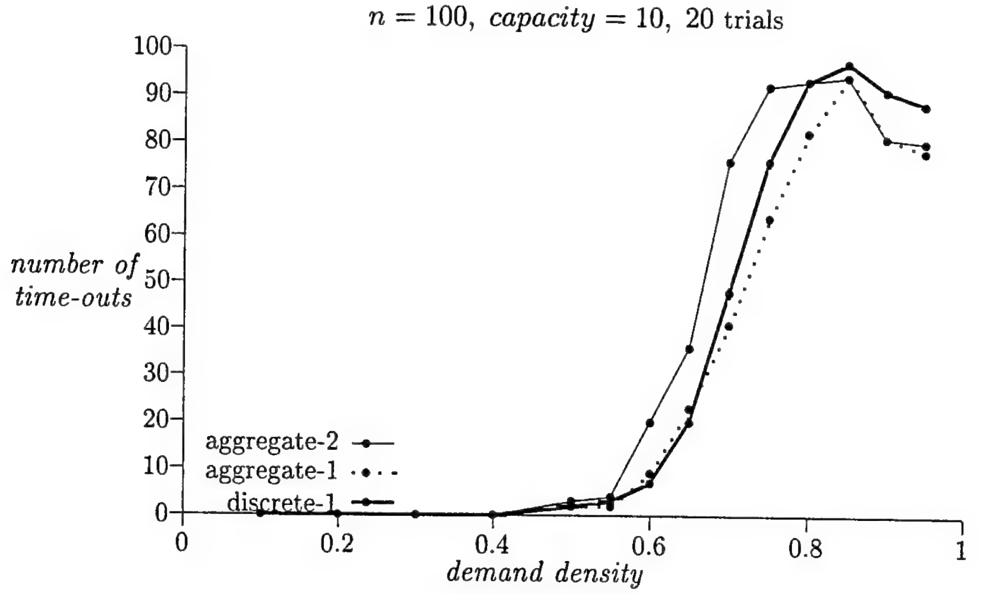

Figure 6: Number of failures (homogeneous processors)

tasks as the problem size.

Generating random problems - heterogeneous processors, no inter-task precedence relations

There are several parameters to be varied in generating random ASR problems in this case:

- c — ASR capacity (10 to 20)
- n — number of tasks (from 50 to 1000)
- *max task duration* — tasks durations are uniformly distributed over the range [1 .. *max-task-duration*].
- *max window* — the width of the *est-lst* time window is uniformly distributed over the range [1 .. *max-window*]
- *demand density* — varies over [0,1]

The task generation strategy was this: generate n tasks of random duration (over the range [1 .. *max-task-duration*]), then sum up the durations to get an aggregate demand *ad*. We know that $ad = c \times td \times dd$ so given c, *ad*, and *dd*, we calculate *td* and then randomly generate earliest start times uniformly over the range [1 .. $td - tsk.duration$] for each task *tsk*. For example, if $c = 20$ and the demand density is $dd = 0.5$, and aggregate demand is $ad = 1000$ then the total duration is $td = 100$.

Experimental results - algorithms Aggregate-1, Aggregate-2, Discrete-1

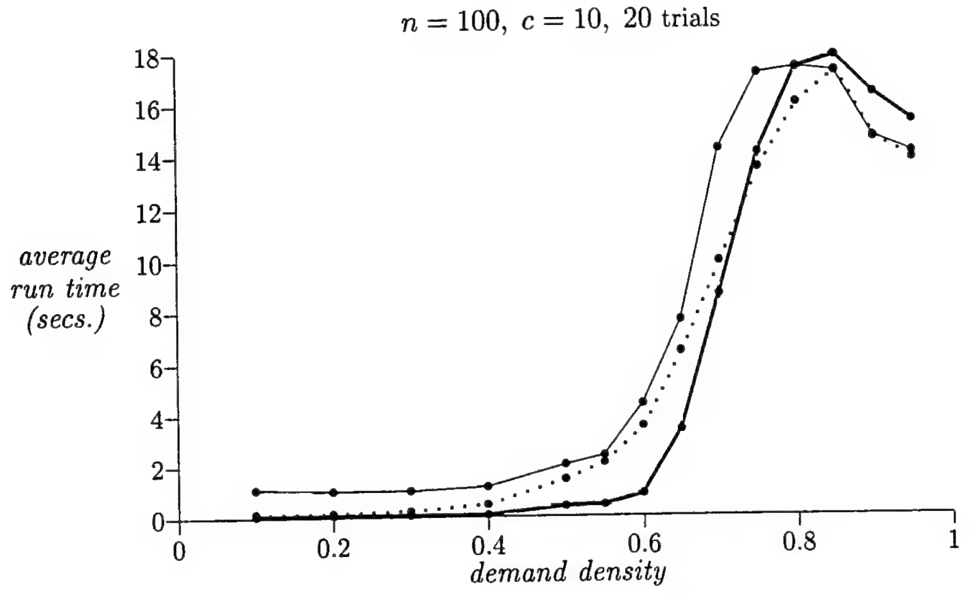

Figure 7: Average case run-time excluding time-outs (homogeneous processors)

The main variable we explored was the average complexity of each scheduling algorithm as a function of demand density. The demand density is the ratio of aggregate task demand to aggregate ASR capacity (capacity times total duration) which we varied from 0 to 1.

As expected, the data shows that the hardest problem instances occur at an intermediate demand density (empirically about 0.8). We wanted to show the performance of the algorithms on reasonably hard task sets, but that entails that some are so hard that they take too long to solve. We introduced a cutoff time of 30 seconds. Figure 6 shows the percentage of task sets that were unsolvable within 30 seconds as function of demand density. Figure 7 compares the average run-times for 3 machine-synthesized ASR algorithms where we excluded from the sample the task sets that timed out.

Generating random problems: heterogeneous processors, inter-task precedence relations

In generating random problems for this variant, additional parameters can be varied:

- processor speeds, for each of k processors
- the partial order on T which expresses the precedence relationship between tasks

A random precedence relationship between tasks was generated by adding predecessors or successors to tasks selected at random from the task set of a random non-precedence problem that was generated as described above. (The probability that an arbitrary task in the original non-precedence problem will be selected is referred to as the *precedence density.*) Note that the complete partial ordering does not need to be reified in the problem structure; it suffices to express the covering relationships that were added by the precedence generation process, since the transitive closure of

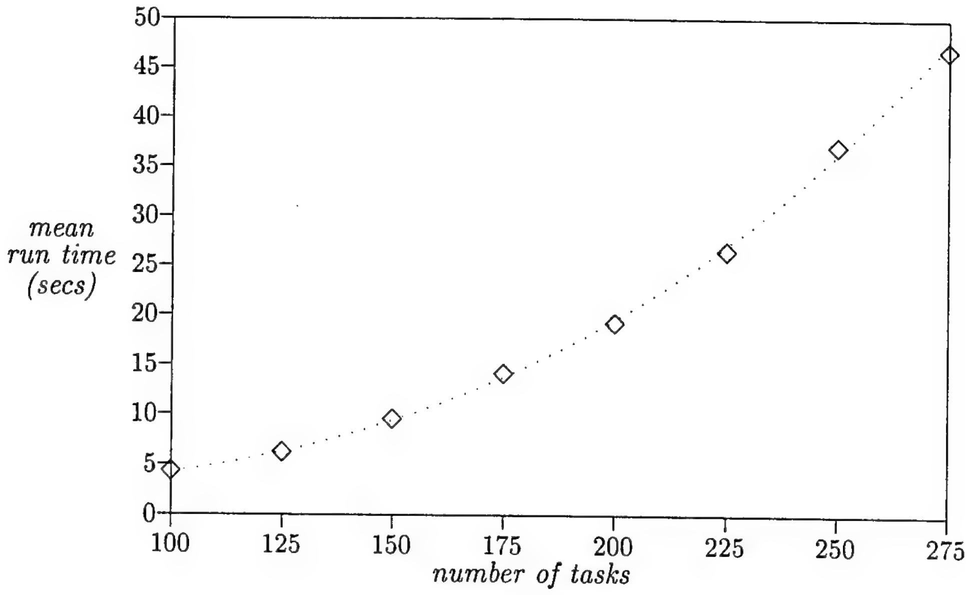

Figure 8: Average case run-time by number of tasks (heterogeneous processors with inter-task precedence relations)

the covering relationship is (in effect) computed by the code that propagates the predecessor and successor constraints.

Experimental results - algorithm Discrete-2

For this set of tests we varied the number of tasks, keeping all other variables constant. Figure 8 shows the rate of increase of time as problem size (expressed as n, the number of tasks in T) increases.

For these runs the other parameters had constant values, as follows:

- k (number of processors): 5
- $speed_i$ (processors speeds): $(2.5, 2.0, 1.5, 1.0, 0.75)$
- *max task duration*: 10
- *max window* : 40
- *precedence density*: 0.25
- *demand density*: 0.3

(The randomly generated task duration and task window variables are uniformly distributed over the ranges [1 .. *max-task-duration*] and [1 .. *max-window*], respectively.)

Curve fitting of the data in Figure 8 shows that it has an $O(n^3)$ rate of growth (the graph shows the data points and the curve $y = 0.0000022 * n^3 + 2$).

5.7. Concluding Remarks

Our overall experience with these algorithms is that they will either find a schedule quickly or else take a very long time to complete. Finding a schedule quickly means that little or no backing up occurs during search – mainly descendants and siblings are ever explored.

From this experience we conclude that the most practical way to exploit these algorithms is to modify them to be nonbacktracking heuristic relaxation algorithms. That is, we modify the control structure to minimize backtracking, and in the case where no descendant of a search tree node survives propagation, then we relax some constraint (typically the latest start time) on a task to the minimum degree necessary to allow the algorithm to proceed. If the task in question has a hard real-time constraint, then backtracking is necessary. This way we build on the strong, efficient constraint propagation techniques and simple control strategy, but obtain an efficient algorithm (usually with low-order polynomial time complexity) that will produce feasible schedules to a slightly relaxed problem instance.

The data presented above does not, of course, provide a complete evaluation of the effectiveness of the algorithms. It would be desirable to assess the tradeoff between completeness, quality of schedule, and runtime, as well as study the effect of varying problem parameters in addition to number of tasks and demand density, such as window size relative to task size, precedence density, number of processors, and (for the heterogeneous case) distribution of total capacity among processors.

6. Limited Discrepancy Search

6.1. Description and Analysis of LDS

Limited Discrepancy Search (LDS) is a variant of global search for traversing a search tree in an order that is likely to find solutions earlier in the traversal than existing strategies such as chronological backtracking. In chronological backtracking, when a failure is encountered, the most-recent choice is undone and an alternative choice made; if that fails then the next-most-recent choice is revisited and so on. Thus alternatives to choices deep in the tree are explored first, before those near the root. However, for many problems the heuristics for making choices improve in their accuracy deeper in the tree where the solution- space is more constrained. Thus the choices near the root of the tree are more likely to be wrong than those deep in the tree. This is not so important if the entire tree can be explored exhaustively, but with reasonable-sized problems this is rarely practical.

The goal of the LDS strategy is to explore paths in the choice tree in the order of likelihood that they will succeed. The LDS strategy first takes the path through the tree given by the best local choice at each choice-point (as does chronological backtracking). Next it considers the paths of discrepancy one: these consist of taking the best local choice at each choice-point except for one choice-point where the second-best choice is made. If the tree is of depth n then there are n such paths. Next paths of discrepancy two are considered and then discrepancy 3 etc. A path of discrepancy two is one where the best choice is taken except at two choice-points where the second-best choice is taken. Harvey and Ginsberg only considered trees with binary choices. For an nary tree one could also classify a path as being of discrepancy two if it consisted of the best choice everywhere but one choice-point where the third-best choice is taken.

There are several options to parallelizing the LDS scheme that trade off communication cost against the cost of redundant computation. To avoid redundant computation one spawns a new process whenever a second-best choice-point is taken. Thus, there is one process initially which always takes the best choice at each choice point, but always spawns a new process with the second-best choice at that point. Thereafter, the new processes always take the best choice. In spawning the new process the complete state of the scheduler must be communicated to the new process, but thereafter the processes are independent except when they are finished they need to communicate to find which has the best solution. The state of the scheduler can become rather large, so the initialization may be significant. Another problem with this parallelization is that initially there is only one process and the number of processes grows linearly in the depth of the tree, so early on there may be many processors idle.

An alternative implementation with minimal communication and idle time is to start with n processes each of which explores the tree from its root, with processor i taking the second- best choice at choice-point i - 1 and the best choice everywhere else. There is one process that always takes the best choice, so for the others, process i is doing redundant work up to choice-point i - 1. If each process takes constant time at each level then up to half of the computation done is redundant. If the computation at level i is O(i) then up to one third of the computation is redundant.

The above analysis assumes one processor per process. In practice there are likely to be significantly fewer processors than the depth of the tree. If we have m processors then we can give each processor the task of computing n/m paths. In this case the redundancy is reduced by approximately the same factor.

6.2. Experiments with LDS

We did experiments to test the effectiveness of LDS using randomly generated scheduling problems. We chose a problem where we had previous experimental experience, that of Asynchronously Shared Resources (ASRs). An ASR is a resource shared simultaneously by many tasks, for example an automobile parking lot having n parking slots.

The main properties we explored were those that varied as a function of demand density. The demand density is the ratio of aggregate task demand to aggregate capacity (capacity times total duration) which we varied from 0 to 1. The greater the demand density the harder it is to find a viable schedule.

There are several parameters used in generating random ASR problems:
c = ASR capacity (20)
n = number of tasks (100)
dmax = tasks durations are uniformly distributed over the range [1 .. 100]
wmax = the width of the earliest to latest start time window is uniformly distributed over the range [1 .. 100]
demand density = varies over [0,1].

In Figure 9 we show the experimental results for how often the scheduler was able to find a schedule for the original method using chronological backtracking and the LDS method with a discrepancy of one. These results clearly show that LDS is effective in finding significantly more solutions when the scheduling problems become difficult.

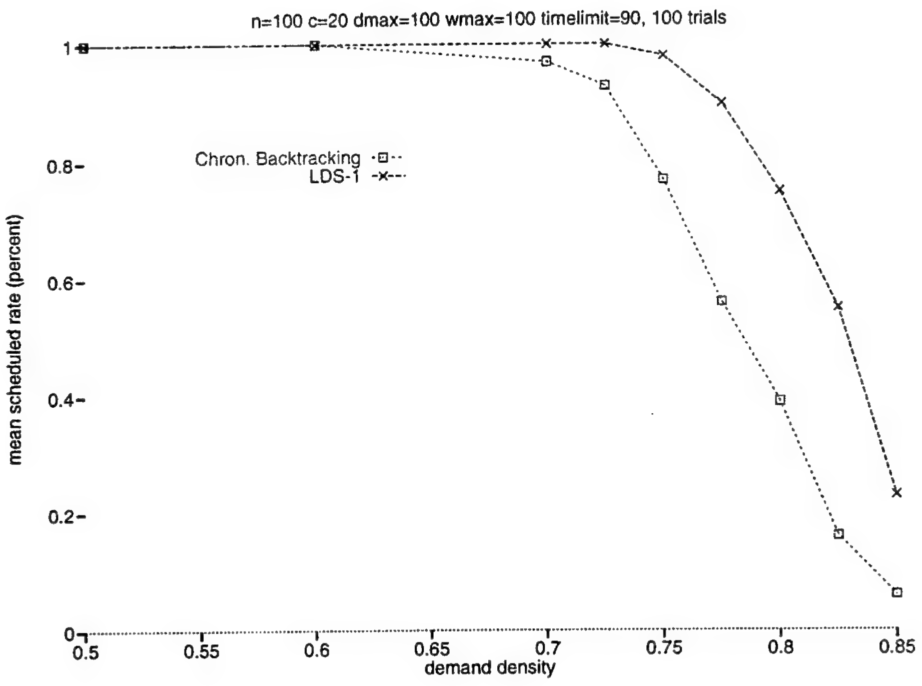

Figure 9: Scheduling Success Rate of Chronological Backtracking vs. LDS

7. Parallel Sorting Algorithms

In this section we apply these ideas to the design of Batcher's Odd-Even sort [1] and discuss the derivation of several other well-known parallel sorting algorithms. Most, if not all, sorting algorithms can be derived as interpretations of the divide-and-conquer paradigm. Accordingly, we present a simplified divide-and-conquer theory and show how it can be applied to design the sort algorithms mentioned above.

7.1. Derivation of a Mergesort

7.1.1. Domain Theory for Sorting

Suppose that we wish to sort a collection of objects belonging to some set α that is linearly-ordered under $\leq$. Here is a simple specification of the sorting problem:

$$\begin{aligned}&\mathit{Sort}(x : \mathit{bag}(\alpha) \mid \mathit{true})\\&\mathit{returns}(\ z : \mathit{seq}(\alpha) \mid x = \mathit{Seq\text{–}to\text{–}bag}(z)\ \wedge\ \mathit{Ordered}(z)\ )\end{aligned}$$

Sort takes a bag (multiset) x of α objects and returns some sequence z such that the following *output condition* holds: the bag of objects in sequence z is the same as x and z must be ordered under $\leq$. The predicate *true* following the parameter x is called the *input condition* and specifies any constraints on inputs.

In order to support this specification formally, we need a domain theory of sorting that includes the theory of sequences and bags, has the linear-order $\langle\alpha, \leq\rangle$ as a parameter, and defines the concepts of *Seq–to–bag* and *Ordered*. The following parameterized theory accomplishes these ends:

Theory $\mathit{Sorting}(\langle\alpha, \leq\rangle : \mathit{linear\text{–}order})$
  Imports $\mathit{integer},\ \mathit{bag}(\alpha), \mathit{seq}(\alpha)$
  Operations
    $\mathit{Ordered} : \mathit{seq}(\alpha) \rightarrow \mathit{Boolean}$

  Axioms

$$\forall(S : \mathit{seq}(\alpha))\ (\mathit{Ordered}(S) \Leftrightarrow \forall(i)(i \in \{1..\mathit{length}(S) - 1\} \implies S(i) \leq S(i+1)))$$

  Theorems

$$\begin{aligned}&\mathit{Ordered}([]) = \mathit{true}\\&\forall(a : \alpha)\ (\mathit{Ordered}([a]) = \mathit{true})\\&\forall(y_1 : \mathit{seq}(\alpha), y_2 : \mathit{seq}(\alpha))\\&\quad(\mathit{Ordered}(y_1 +\!\!+ y_2) \Leftrightarrow \mathit{Ordered}(y_1)\\&\qquad\wedge\ \mathit{Seq\text{–}to\text{–}bag}(y_1) \leq \mathit{Seq\text{–}to\text{–}bag}(y_2)\\&\qquad\wedge\ \mathit{Ordered}(y_2))\end{aligned}$$

  end–theory

Sorting theory imports *integer*, *bag*, and *sequence* theory. Sequences are constructed via $[]$ (empty sequence), $[a]$ (singleton sequence), and $A{+\!\!+}B$ (concatenation). For example,

$$[1,2,3]{+\!\!+}[4,5,6] = [1,2,3,4,5,6].$$

Several parallel sorting algorithms are based on an alternative set of constructors which use interleaving in place of concatenation: the *ilv* operator

$$[1,2,3] \ ilv \ [4,5,6] = [1,4,2,5,3,6]$$

interleaves the elements of its arguments. We assume that the arguments to *ilv* have the same length, typically denoted n, and that it is defined by

$$A \ ilv \ B = C \Leftrightarrow \forall(i)(i \in \{1..n\} \implies C_{2i-1} = A_i \ \wedge \ C_{2i} = B_i).$$

In Section 7.2. we develop some of the theory of sequences based on the *ilv* constructor.

Bags have an analogous set of constructors: $\{\!\{\}\!\}$ (empty bag), $\{\!\{a\}\!\}$ (singleton bag), and $A \uplus B$ (associative and commutative bag union). The operator *Seq-to-bag* coerces sequences to bags by forgetting the ordering implicit in the sequence. *Seq-to-bag* obeys the following distributive laws:

$$Seq\text{–}to\text{–}bag([]) = \{\!\{\}\!\}$$

$$\forall(a : \alpha) \ Seq\text{–}to\text{–}bag([a]) = \{\!\{a\}\!\}$$

$$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$$
$$Seq\text{–}to\text{–}bag(y_1{+\!\!+}y_2) = Seq\text{–}to\text{–}bag(y_1) \uplus Seq\text{–}to\text{–}bag(y_1)$$

$$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$$
$$Seq\text{–}to\text{–}bag(y_1 \ ilv \ y_2) = Seq\text{–}to\text{–}bag(y_1) \uplus Seq\text{–}to\text{–}bag(y_1)$$

In the sequel we will omit universal quantifiers whenever it is possible to simplify the presentation without sacrificing clarity.

7.1.2. Divide-and-Conquer Theory

Most sorting algorithms are based on the divide-and-conquer paradigm: If the input is primitive then a solution is obtained directly, by simple code. Otherwise a solution is obtained by decomposing the input into parts, independently solving the parts, then composing the results. Program termination is guaranteed by requiring that decomposition is monotonic with respect to a suitable well-founded ordering. In this paper we focus on divide-and-conquer algorithms that have the following general form:

```
$DC(x_0 : D \mid I(x_0))$
$returns(\ z : R \mid O(x_0, z))$
$=$ **if** $Primitive(x_0)$
    **then** $Directly\text{–}Solve(x_0)$
    **else** $let\ \langle x_1, x_2\rangle = Decompose(x_0)$
        $Compose(DC(x_1),\ DC(x_2))$
```

We refer to *Decompose* as a decomposition operator, *Compose* as a composition operator, *Primitive* as a control predicate, and *Directly–Solve* as a primitive operator.

The essence of a divide-and-conquer algorithm can be presented via a *reduction diagram*:

$$\begin{tikzcd} x_0 \arrow[r, "DC"] \arrow[d, "Decompose"'] & z_0 \\ \langle x_1, x_2 \rangle \arrow[r, "DC \times DC"] & \langle z_1, z_2 \rangle \arrow[u, "Compose"'] \end{tikzcd}$$

which should be read as follows. Given input x_0, an acceptable solution z_0 can be found by decomposing x_0 into two subproblems x_1 and x_2, solving these subproblem recursively yielding solutions z_1 and z_2 respectively, and then composing z_1 and z_2 to form z_0.

In the derivations of this paper we will usually ignore the primitive predicate and *Directly-Solve* operator – the interesting design work lies in calculating compatible pairs of *Decompose* and *Compose* operators.

The following mergesort program is an instance of this scheme:

```
$MSort(b_0 : bag(integer))$
    $returns(\ z : seq(\alpha) \mid x = Seq\text{–}to\text{–}bag(z) \ \wedge \ Ordered(z)\ )$
    = **if** $size(b_0) \leq 1$
        **then** $b_0$
        **else** $let\ \langle b_1, b_2 \rangle = Split(b_0)$
                $Merge(MSort(b_1),\ MSort(b_2))$
```

Here *Split* decomposes a bag into two subbags of roughly equal size and *Merge* composes two sorted sequences to form a sorted sequence.

The characteristic that subproblems are solved independently gives the divide-and-conquer notion its great potential in parallel environments. Another aspect of divide-and-conquer is that the recursive decomposition can often be performed implicitly, thereby enabling a purely bottom-up computation. For example, in the Mergesort algorithm, the only reason for the recursive splitting is to control the order of composition (merging) of sorted subproblem solutions. However the pattern of merging is easily determined at design-time and leads to the usual binary tree computation pattern.

To express the essence of divide-and-conquer, we define a divide-and-conquer theory comprised of various sorts, function, predicates, and axioms that assure that the above scheme correctly solves a given problem. A simplified divide-and-conquer theory is as follows (for more details see [18, 21]):

Theory *Divide–and–Conquer*
  Sorts D, R domain and range of a problem
  Operations
    $I : D \rightarrow Boolean$ input condition

$O : D \times R \rightarrow Boolean$	output condition
$primitive : D \rightarrow Boolean$	control predicate
$O_{Decompose} : D \times D \times D \rightarrow Boolean$	output condition for Decompose
$O_{Compose} : R \times R \times R \rightarrow Boolean$	output condition for Compose
$\succ : D \times D \rightarrow Boolean$	well-founded order

Soundness Axiom

$$
\begin{array}{l}
O_{Decompose}(x_0, x_1, x_2) \\
\ \wedge O(x_1, z_1) \ \wedge \ O(x_2, z_2) \\
\ \wedge O_{Compose}(z_0, z_1, z_2) \\
\ \Longrightarrow \ O(x_0, z_0)
\end{array}
$$

...

end–theory

The intuitive meaning of the Soundness Axiom is that if input x_0 decomposes into a pair of subproblems $\langle x_1, x_2 \rangle$, and z_1 and z_2 are solutions to subproblems x_1 and x_2 respectively, and furthermore solutions z_1 and z_2 can be composed to form solution z_0, then z_0 is guaranteed to be a solution to input x_0. There are other axioms that are required: well-foundedness conditions on $\succ$ and *admissibility conditions* that assure that Decompose and Compose can be refined to total functions over their domains. We ignore these in order to concentrate on the essentials of the design process.

The main difficulty in designing an instance of the divide-and-conquer scheme for a particular problem lies in constructing decomposition and composition operators that work together. The following is a simplified version of a tactic in [18].

1. Choose a simple decomposition operator and well-founded order.
2. Derive the control predicate based on the conditions under which the decomposition operator preserves the well-founded order and produces legal subproblems.
3. Derive the input and output conditions of the composition operator using the Soundness Axiom of divide-and-conquer theory.
4. Design an algorithm for the composition operator.
5. Design an algorithm for the primitive operator.

Mergesort is derived by choosing $\uplus^{-1}$ as a simple (nondeterministic) decomposition operator. A specification for the well-known merge operation is derived using the Soundness Axiom.

$$\begin{tikzcd}
b_0 \arrow[r, "Sort"] \arrow[d, "\uplus^{-1}"'] & z_0 \\
<b_1, b_2> \arrow[r, "Sort \times Sort"] & <z_1, z_2> \arrow[u, "Merge"']
\end{tikzcd}$$

A similar tactic based on choosing a simple composition operator and then solving for the decomposition operator is also presented in [18]. This tactic can be used to derive selection sort and quicksort-like algorithms.

Deriving the output condition of the composition operator is the most challenging step and bears further explanation. The *Soundness Axiom* of divide-and-conquer theory relates the output conditions of the subalgorithms to the output condition of the whole divide-and-conquer algorithm:

$$\begin{aligned}
&O_{Decompose}(x_0, x_1, x_2)\\
&\wedge O(x_1, z_1) \ \wedge \ O(x_2, z_2)\\
&\wedge O_{Compose}(z_0, z_1, z_2)\\
&\implies O(x_0, z_0)
\end{aligned}$$

For design purposes this constraint can be treated as having three unknowns: O, $O_{Decompose}$, and $O_{Compose}$. Given O from the original specification, we supply an expression for $O_{Decompose}$ then reason backwards from the consequent to an expression over the program variables z_0, z_1, and z_2. This derived expression is taken as the output condition of *Compose*.

Returning to Mergesort, suppose that we choose $\uplus^{-1}$ as a simple decomposition operator. To proceed with the tactic, we instantiate the Soundness Axiom with the following substitutions

$$\begin{aligned}
O_{Decompose} &\mapsto \lambda(b_0, b_1, b_2)\ b_0 = b_1 \uplus b_2\\
O &\mapsto \lambda(b, z)\ b = \mathit{Seq\text{--}to\text{--}bag}(z) \ \wedge \ \mathit{Ordered}(z)
\end{aligned}$$

yielding

$$\begin{aligned}
&b_0 = b_1 \uplus b_2\\
&\wedge b_1 = \mathit{Seq\text{--}to\text{--}bag}(z_1) \ \wedge \ \mathit{Ordered}(z_1)\\
&\wedge b_2 = \mathit{Seq\text{--}to\text{--}bag}(z_2) \ \wedge \ \mathit{Ordered}(z_2)\\
&\wedge O_{Compose}(z_0, z_1, z_2)\\
&\implies b_0 = \mathit{Seq\text{--}to\text{--}bag}(z_0) \ \wedge \ \mathit{Ordered}(z_0)
\end{aligned}$$

To derive $O_{Compose}(z_0, z_1, z_2)$ we reason backwards from the consequent $b_0 = \mathit{Seq\text{--}to\text{--}bag}(z_0) \ \wedge$ $\mathit{Ordered}(z_0)$ toward a sufficient condition expressed over the variables $\{z_0, z_1, z_2\}$ modulo the assumptions of the antecedent:

$$b_0 = \mathit{Seq\text{--}to\text{--}bag}(z_0) \ \wedge \ \mathit{Ordered}(z_0)$$

$$\Longleftrightarrow \qquad \text{using assumption } b_0 = b_1 \uplus b_2$$

$$b_1 \uplus b_2 = \mathit{Seq\text{--}to\text{--}bag}(z_0) \ \wedge \ \mathit{Ordered}(z_0)$$

$$\Longleftrightarrow \qquad \text{using assumption } b_i = \mathit{Seq\text{--}to\text{--}bag}(z_i), \ i = 1, 2$$

$$\begin{aligned}
&\mathit{Seq\text{--}to\text{--}bag}(z_1) \uplus \mathit{Seq\text{--}to\text{--}bag}(z_2) = \mathit{Seq\text{--}to\text{--}bag}(z_0)\\
&\wedge \mathit{Ordered}(z_0).
\end{aligned}$$

This last expression is a sufficient condition expressed in terms of the variables $\{z_0, z_1, z_2\}$ and so we take it to be the output condition for *Compose*. In other words, we ensure that the Soundness Axiom holds by taking this expression as a constraint on the behavior of the composition operator.

The input condition to the composition operator is obtained by forward inference from the antecedent of the soundness axiom; here we have the (trivial) consequences $Ordered(z_1)$ and $Ordered(z_2)$. Only consequences expressed in terms of the input variables z_1 and z_2 are useful.

Thus we have derived a formal specification for *Compose*:

$Merge(A : seq(integer), B : seq(integer) \mid Ordered(A) \ \wedge \ Ordered(B))$
$\quad returns(\ z : seq(integer)$
$\qquad\qquad \mid Seq\text{--}to\text{--}bag(A) \uplus Seq\text{--}to\text{--}bag(B) = Seq\text{--}to\text{--}bag(z)$
$\qquad\qquad \wedge \ Ordered(z) \ ).$

Merge is now a derived concept in *Sorting* theory. We later derive laws for it, but now we proceed to design an algorithm to satisfy this specification. The usual sequential algorithm for merging is based on choosing a simple "cons" composition operator and deriving a decomposition operator [18]. However this algorithm is inherently sequential and requires linear time.

7.2. Batcher's Odd-Even Sort

Batcher's Odd-Even sort algorithm [1] is a mergesort algorithm in which the merge operator itself is a divide-and-conquer algorithm. The Odd-Even merge is derived by choosing a simple decomposition operator based on *ilv* and deriving constraints on the composition operator.

Before proceeding with algorithm design we need to develop some of the theory of sequences based on the *ilv* constructor. Generally, we develop a domain theory by deriving laws about the various concepts of the domain. In particular we have found that distributive, monotonicity, and invariance laws provide most of the laws needed to support formal design. This suggests that we develop laws for various sorting concepts, such as *Seq-to-bag* and *Ordered*. From Section 7.1. we have

Theorem 1. *Distributing Seq-to-bag over sequence constructors.*

1.1. $Seq\text{--}to\text{--}bag([]) = \{\!\{\}\!\}$
1.2. $Seq\text{--}to\text{--}bag([a]) = \{\!\{a\}\!\}$
1.3. $Seq\text{--}to\text{--}bag(S_1 \ ilv \ S_2) = Seq\text{--}to\text{--}bag(S_1) \uplus Seq\text{--}to\text{--}bag(S_2)$

It is not obvious how to distribute *Ordered* over *ilv* , so we try to derive it. In this derivation let n denote the length of both A and B.

$Ordered(A \ ilv \ B)$

$\iff$ by definition of *Ordered*

$$\forall(i)(i \in \{1..2n-1\} \implies (A\ ilv\ B)_i \leq (A\ ilv\ B)_{i+1})$$

$\iff$ change of index

$$\forall(j)(j \in \{1..n\} \implies (A\ ilv\ B)_{2j-1} \leq (A\ ilv\ B)_{2j})$$
$$\wedge\ \forall(j)(j \in \{1..n-1\} \implies (A\ ilv\ B)_{2j} \leq (A\ ilv\ B)_{2j+1})$$

$\iff$ by definition of *ilv*

$$\forall(j)(j \in \{1..n\} \implies A_j \leq B_j)$$
$$\wedge\ \forall(j)(j \in \{1..n-1\} \implies B_j \leq A_{j+1}).$$

These last two conjuncts are similar in form and suggest the need for a new concept definition and perhaps new notation. Suppose we define $A \leq^* B$ *iff* $A_i \leq B_i$ for $i \in \{1 \ldots n\}$. This allows us to express the first conjunct as $A \leq^* B$, but then we cannot quite express the second concept – we need to generalize to allow an offset in the comparison:

Definition 1. A pair of sequences A and B of length n are *pairwise-ordered with offset k*, written $A \leq^*_k B$, iff $A_i \leq B_{i+k}$ for $i \in \{1 \ldots n-k\}$.

Then the derivation above yields the following simple law

Theorem 2. *Conditions under which an interleaved sequence is Ordered.*
For all sequences A, B,
$Ordered(A\ ilv\ B) \iff A \leq^*_0 B \ \wedge\ B \leq^*_1 A.$

Note that this definition provides a proper generalization of the notion of orderedness:

Theorem 3. *Ordered as a diagonal specialization of* $\leq^*_i$.
For all sequences S,
$Ordered(S) \iff S \leq^*_1 S$

Other laws are easily derived:

Theorem 4. *Transitivity of* $\leq^*_i$.
For all sequences A, B, C of equal length and integers i and j,
$A \leq^*_i B \ \wedge\ B \leq^*_j C \implies A \leq^*_{i+j} C$

As a simple consequence we have

Corollary 1. *Only Ordered sequences interleave to form Ordered sequences.*
For all sequences A, B,
$Ordered(A\ ilv\ B) \implies Ordered(A) \ \wedge\ Ordered(B).$

Proof:

Ordered(A ilv B)

$\iff$ by Theorem 2

$A \leq_0^* B \wedge B \leq_1^* A$

$\implies$ applying Theorem 4 twice

$A \leq_1^* A \wedge B \leq_1^* B$

$\iff$ by Theorem 3

$Ordered(A) \wedge Ordered(B)$. □

Theorem 5. *Monotonicity of $\leq_i^*$ with respect to merging.*
For all sequences A_1, A_2, B_1, and B_2 and integers i,
$A_1 \leq_i^* A_2 \wedge B_1 \leq_i^* B_2 \implies Merge(A_1, B_1) \leq_{2i}^* Merge(A_2, B_2)$

We can apply the basic sort operation $sort2(x, y) = \langle min(x, y), max(x, y)\rangle$ over parallel sequences, just as we did with the comparator $\leq$.

Definition 2. *Pairwise-sort of sequences with offset k.*
Define $sort2_k^*(A, B) = \langle A', B'\rangle$ such that
(1) for $i \leq k$, $B'_i = B_i$
(2) for $i = 1, \ldots, n-k$, $\langle A'_i, B'_{i+k}\rangle = sort2(A_i, B_{i+k})$
(3) for $i > n-k$, $A'_i = A_i$

For example, $sort2_1^*([2,3,8,9],[0,1,4,5]) = \langle[1,3,5,9],[0,2,4,8]\rangle$. Laws for $sort2_k^*$ can be developed:

Theorem 6. *$sort2_k^*$ establishes $\leq_k^*$.*
For all sequences A, B, A', and B', and integer k,
$sort2_k^*(A, B) = \langle A', B'\rangle \implies A' \leq_k^* B'$.

This theorem is a trivial consequence of the definition of $sort2_k^*$. The following theorems give conditions under which important properties of the domain theory ($\leq_i^*$, *Ordered*) are preserved under under the $sort2_k^*$ operation. They can be proved using straightforward analysis of cases.

Theorem 7. *Ordered is invariant under $sort2_k^*$.*
For all sequences A, B and integer k,
$Ordered(A) \wedge Ordered(B) \wedge sort2_k^*(A, B) = \langle A', B'\rangle$
$\implies Ordered(A') \wedge Ordered(B')$

Theorem 8. *Invariance of $A \leq_i^* B$ with respect to $sort2_k^*(A,B)$.*
For all sequences A, B and integers i and k,
$A \leq_i^* B \land sort2_k^*(A,B) = \langle A', B' \rangle \implies A' \leq_i^* B'$

Theorem 9. *Invariance of $A \leq_i^* B$ with respect to $sort2_k^*(B,A)$.*
For all sequences A, B and $0 \leq i < k$,
$A \leq_{i+k}^* A \land B \leq_{i+k}^* B \land A \leq_i^* B \land B \leq_{i+2k}^* A \land sort2_k^*(B,A) = \langle B', A' \rangle$
$\implies A' \leq_i^* B'$

With these concepts and laws in hand, we can proceed to derive Batcher's Odd-Even mergesort. It can be derived simply by choosing to decompose the inputs to Merge by uninterleaving them.

$$\begin{tikzcd}
\langle A_0, B_0 \rangle \arrow[r, "Merge"] \arrow[d, "ilv^{-2}"'] & S_0 : seq(integer) \\
\langle \langle A_1, B_1 \rangle, \langle A_2, B_2 \rangle \rangle \arrow[r, "Merge \times Merge"] & \langle S_1, S_2 \rangle \arrow[u, "?"']
\end{tikzcd}$$

where ilv^{-2} means $A_0 = A_1 \; ilv \; A_2$ and $B_0 = B_1 \; ilv \; B_2$. Note how this decomposition operator creates subproblems of roughly the same size which provides good opportunities for parallel computation. Note also that this decomposition operator must ensure that the subproblems $\langle A_1, B_1 \rangle$ and $\langle A_2, B_2 \rangle$ satisfy the input conditions of *Merge*. This property is assured by Corollary 1.

We proceed by instantiating the Soundness Axiom as before:

$$\begin{aligned}
& A_0 = A_1 \; ilv \; A_2 \land Ordered(A_0) \\
& \land B_0 = B_1 \; ilv \; B_2 \land Ordered(B_0) \\
& \land Seq\text{–}to\text{–}bag(S_1) = Seq\text{–}to\text{–}bag(A_1) \uplus Seq\text{–}to\text{–}bag(B_2) \land Ordered(S_1) \\
& \land Seq\text{–}to\text{–}bag(S_2) = Seq\text{–}to\text{–}bag(A_2) \uplus Seq\text{–}to\text{–}bag(B_2) \land Ordered(S_2) \\
& \land O_{Compose}(S_0, S_1, S_2) \\
& \implies Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_0) \uplus Seq\text{–}to\text{–}bag(B_0) \\
& \qquad \land Ordered(S_0)
\end{aligned}$$

Constraints on $O_{Compose}$ are derived as follows:

$$Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_0) \uplus Seq\text{–}to\text{–}bag(B_0) \land Ordered(S_0)$$

$\Longleftrightarrow$ by assumption

$$Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_1 \; ilv \; A_2) \\ \uplus \; Seq\text{–}to\text{–}bag(B_1 \; ilv \; B_2)$$

$$\wedge \; Ordered(S_0)$$

$\Longleftrightarrow$ distributing *Seq–to–bag* over *ilv*

$$Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(A_1) \uplus Seq\text{–}to\text{–}bag(A_2) \\ \uplus \; Seq\text{–}to\text{–}bag(B_1) \uplus Seq\text{–}to\text{–}bag(B_2) \\ \wedge \; Ordered(S_0)$$

$\Longleftrightarrow$ by assumption

$$Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(S_1) \uplus Seq\text{–}to\text{–}bag(S_2) \\ \wedge \; Ordered(S_0).$$

The input conditions on *Merge* are derived by forward inference from the assumptions above:

$$A_0 = A_1 \; ilv \; A_2 \; \wedge \; Ordered(A_0)$$

$$\wedge \; B_0 = B_1 \; ilv \; B_2 \; \wedge \; Ordered(B_0)$$

$$\wedge \; Ordered(S_1) \; \wedge \; Ordered(S_2)$$

$\Longrightarrow$ distributing *Ordered* over *ilv*

$$A_1 \leq_0^* A_2 \; \wedge \; A_2 \leq_1^* A_1 \\ \wedge \; B_1 \leq_0^* B_2 \; \wedge \; B_2 \leq_1^* B_1 \\ \wedge \; Ordered(S_1) \; \wedge \; Ordered(S_2)$$

$\Longrightarrow$ by monotonicity of $\leq_i^*$ with respect to *Merge*

$$S_1 \leq_0^* S_2 \; \wedge \; S_2 \leq_2^* S_1 \\ \wedge \; Ordered(S_1) \; \wedge \; Ordered(S_2).$$

Thus we have derived the specification

$$Merge\text{–}Compose(S_1 : seq(integer), S_2 : seq(integer) \\ \mid S_1 \leq_0^* S_2 \; \wedge \; S_2 \leq_2^* S_1 \; \wedge \; Ordered(S_1) \; \wedge \; Ordered(S_2)) \\ returns(\; S_0 : seq(integer) \\ \mid Seq\text{–}to\text{–}bag(S_0) = Seq\text{–}to\text{–}bag(S_1) \uplus Seq\text{–}to\text{–}bag(S_2) \\ \wedge \; Ordered(S_0) \;).$$

How can this specification be satisfied? Theorems 1.3 and 2 suggest *ilv* since it would establish the output conditions of *Merge–Compose*. Theorem 2 requires that we achieve the input condition $S_1 \leq_0^* S_2 \wedge S_2 \leq_1^* S_1$ first. But Theorem 6 ($sort2_k^*$ establishes $\leq_k^*$) enables us to apply $Sort2_1^*(S_2, S_1)$ in order to achieve the second conjunct. Theorems 7, 8, and 9 ensure that $S_1 \leq_0^* S_2$ remains invariant. So *Merge–Compose* is satisfied by $\tilde{ilv} \cdot sort2_1^*(S_2, S_1)$. The final algorithm in diagram form is

$$\begin{tikzcd}
b_0 \arrow[r, "Sort"] \arrow[d, "\uplus^{-1}"'] & z_0 \\
\langle b_1, b_2 \rangle \arrow[r, "Sort \times Sort"] & \langle z_1, z_2 \rangle \arrow[u, "Merge"']
\end{tikzcd}$$

$$\begin{tikzcd}
\langle A_0, B_0 \rangle \arrow[r, "Merge"] \arrow[d, "ilv^{-2}"'] & S_0 \\
\langle \langle A_1, B_1 \rangle, \langle A_2, B_2 \rangle \rangle \arrow[r, "Merge \times Merge"] & < S_1, S_2 > \arrow[u, "\tilde{ilv} \cdot sort2_1^*(S_2, S_1)"']
\end{tikzcd}$$

To simplify the analysis, assume that the input to *Sort* has length $n = 2^m$. Given n processors, *Merge* runs in time

$$\begin{aligned} T_{Merge}(n) &= max(T_{Merge}(n/2), T_{Merge}(n/2)) + O(1) \\ &= O(log(n)) \end{aligned}$$

since the decomposition and composition operators both can be evaluated in constant time and the recursion goes to depth $O(log(n))$.

The decomposition operator $\uplus^{-1}$ in *Sort* is nondeterministic. This is an advantage at this stage of design since it allows us to defer commitments and make choices that will maximize performance. In this case the complexity of *Sort* is calculated via the recurrence

$$T_{Sort}(n) = max(T_{Sort}(a(n)), T_{Sort}(b(n))) + O(log(n))$$

which is optimized by taking $a(n) = b(n) = n/2$ – that is, we split the input bag in half. Given n processors this algorithm runs in $O(log^2(n))$ time, so it is suboptimal for sorting. However, according to [13], Batcher's Odd-Even sort is the most commonly used of parallel sort algorithms.

7.3. Related Sorting Algorithms

Several other parallel sorting algorithms can be developed using the techniques above. Batcher's bitonic sort [1] and the Periodic Balanced Sort [6] are also basically mergesort algorithms. They differ from Odd-Even sort in that the merge operation is a divide-and-conquer based on concatenation as the composition operator. For example, bitonic merge can be diagrammed as follows:

$$\begin{tikzcd}
\langle A, B \rangle \arrow[r, "Merge"] \arrow[d, "{[id, reverse]}"'] & S_0 \\
\langle A_0, B_0 \rangle \arrow[r, "BMerge"] \arrow[d, "{[halve, halve] \cdot sort2_0^*}"'] & S_0 \arrow[u, "id"'] \\
\langle \langle A_1, B_1 \rangle, \langle A_2, B_2 \rangle \rangle \arrow[r, "BMerge \times BMerge"] & < S_1, S_2 > \arrow[u, "+\!\!+"']
\end{tikzcd}$$

The essential fact about using $+\!\!+$ as a composition operator is that $\langle\langle A_1, B_1\rangle,$ $\langle A_2, B_2\rangle\rangle$ must be a partition in the sense that no element of A_1 or B_1 is greater than any element of A_2 and B_2. The cleverness of the algorithm lies in a special property of sequences that allows a simple operation ($sort2_0^*$ here) to effectively produce a partition. This property is called "bitonicity" for bitonic sort and "balanced" for the periodic balanced sort. (The operation $\langle A_0, B_0\rangle = \langle id(A), reverse(B)\rangle$ establishes the bitonic property and decomposition preserves it). The challenge in deriving these algorithms lies in discovering these properties given that one wants a divide-and-conquer algorithm based on $+\!\!+$ as composition. Is there a systematic way to discover these properties or must we rely on creative invention? Admittedly, there may be other frameworks within which the discovery of these properties is easier.

Another well-known parallel sort algorithm is odd-even transposition sort. This can be viewed as a parallel variant of bubble-sort which in turn is derivable as a selection sort (local search is used to derive the selection subalgorithm).

The *ilv* constructor for sequences has many other applications including polynomial evaluation, discrete fast fourier transform, and matrix transposition. Butterfly and shuffle networks are natural architectures for implementing algorithms based on *ilv* [12].

7.4. Concluding Remarks

The Odd-Even sort algorithm is simpler to state than to derive. The properties of a *ilv*-based theory of sequences are much harder to understand and develop than a concatenation-based theory. However, the payoff is an abundance of algorithms with good parallel properties.

8. Concluding Remarks

The applications described above span a range of ways in which synthesis technology can support parallel software engineering. The Batcher's sort example illustrates the synthesis of a well-known high-performance sorting algorithm. The scheduler of real-time concurrent tasks illustrates the use of synthesis thechnology to support *meta-synthesis*: we synthesize a program (scheduler) that generates a program (the schedule that actually executes on the parallel hardware). The final example of LDS illustrates how parallel search strategy knowledge can be formally captured and applied – furthermore we have applied it to the parallization of a meta-synthesizer by parallelizing the scheduler of real-time parallel processes.

References

[1] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[2] Eric Biefeld and Lynne Cooper. Operations mission planner. Technical Report JPL 90-16, Jet Propulsion Laboratory, March 1990.

[3] Mark S. Boddy. Temporal reasoning for planning and scheduling in complex domains: Lessons learned. In Austin Tate, editor, *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*, pages 77–83. AAAI Press, Menlo Park, CA, 1996.

[4] M.B. Burstein and D.R. Smith. ITAS: A portable interactive transportation scheduling tool using a search engine generated from formal specifications. In *Proceedings of the Third International Conference on Artificial Intelligence Planning System (AIPS-96)*, Edinburgh, UK, May 1996.

[5] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1989.

[6] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *Journal of the ACM*, 36(4):738–757, October 1989.

[7] Mark S. Fox, Norman Sadeh, and Can Baykan. Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 309–315, Detroit, MI, August 20–25, 1989.

[8] Mark S. Fox and Stephen F. Smith. ISIS – a knowledge-based system for factory scheduling. *Expert Systems*, 1(1):25–49, July 1984.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness.* W.H. Freeman, San Francisco, CA, 1979.

[10] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.

[11] Leslie A. Hall. Approximation algorithms for scheduling. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 1–45. PWS Publishing Company, Boston, MA, 1997.

[12] Geraint Jones and Mary Sheeran. Collecting butterflies. Technical Report PRG-91, Oxford University, Programming Research Group, February 1991.

[13] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[14] Steven Minton, Mark Johnson, Andrew B. Philips, and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conferenceon Artificial Intelligence*, pages 290–295, 1990.

[15] W.P.M. Nuijten. *Time and Resource Constrained Scheduling.* PhD thesis, Eindhoven University, The Netherlands, 1994.

[16] Norman Sadeh. Look-ahead techniques for micro-opportunistic job shop scheduling. Technical Report CMU-CS-91-102, Carnegie-Mellon University, March 1991.

[17] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conferenceon Artificial Intelligence*, pages 440–446, 1992.

[18] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, September 1985. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[19] Douglas R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, November 1987.

[20] Douglas R. Smith. KIDS: A knowledge-based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, Menlo Park, 1991.

[21] Douglas R. Smith. Structure and design of problem reduction generators. In B. Möller, editor, *Constructing Programs from Specifications*, pages 91–124. North-Holland, Amsterdam, 1991.

[22] Douglas R. Smith. Classification approach to design. Technical Report KES.U.93.4, Kestrel Institute, 1993.

[23] Douglas R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming*, 15(5-6):571–606, May-June 1993.

[24] Douglas R. Smith. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering*, 16(9):1024–1043, September 1990.

[25] Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2-3):305–321, October 1990.

[26] Douglas R. Smith and Stephen J. Westfold. Scheduling an asynchronous shared resource. Technical report, Kestrel Institute, February 1996.

[27] Stephen F. Smith. The OPIS framework for modeling manufacturing systems. Technical Report CMU-RI-TR-89-30, The Robotics Institute, Carnegie-Mellon University, December 1989.

[28] Jia Xu and David Lorge Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion. *IEEE Transactions on Software Engineering*, SE-16(3):360–369, March 1990.

[29] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard real-time systems. In *Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 132–146, New Orleans, LA, December 4-6, 1991. ACM SIGSOFT. Also in: Software Engineering Notes, Vol. 16(5), December 1991.

[30] W. Zhao, K. Ramamritham, and J. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, SE-12(5):564–577, May 1987.

[31] Monte Zweben, Michael Deale, and Robert Gargan. Anytime rescheduling. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 215–219, San Diego, CA, November 5–8, 1990. DARPA.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61156